Part I

2. Types and Basic Design Methods for Randomized Algorithms

Chapter 2. DESIGN METHODS for RANDOMIZED ALGORITHMS

Chapter 2. DESIGN METHODS for RANDOMIZED ALGORITHMS

In this chapter:

1. Main types of randomized algorithms are introduced and illustrated.

Chapter 2. DESIGN METHODS for RANDOMIZED ALGORITHMS

In this chapter:

1. Main types of randomized algorithms are introduced and illustrated.
2. Main design methods for randomized algorithms are introduced and illustrated.

BASIC PROBABILITIES FOR RANDOMIZED ALGORITHMS

BASIC PROBABILITIES FOR RANDOMIZED ALGORITHMS

Let us consider any decision problem P.

BASIC PROBABILITIES FOR RANDOMIZED ALGORITHMS

Let us consider any decision problem P.

- For any randomized algorithm A for P, and for any input x of A, let the set $S_{A,x}$ of all runs of A on x be the main sample space for the analysis of A on the input x.

BASIC PROBABILITIES FOR RANDOMIZED ALGORITHMS

Let us consider any decision problem P.

- For any randomized algorithm A for P, and for any input x of A, let the set $S_{A,x}$ of all runs of A on x be the main sample space for the analysis of A on the input x.
- If one chooses the random variable $V_{A,x}$ that assigns 1 (0) to any run of A on x with the correct (wrong) output, then the expectation value of $V_{A,x}$ is exactly the success probability of A on x.

BASIC PROBABILITIES FOR RANDOMIZED ALGORITHMS

Let us consider any decision problem P.

- For any randomized algorithm A for P, and for any input x of A, let the set $S_{A,x}$ of all runs of A on x be the main sample space for the analysis of A on the input x.
- If one chooses the random variable $V_{A,x}$ that assigns 1 (0) to any run of A on x with the correct (wrong) output, then the expectation value of $V_{A,x}$ is exactly the success probability of A on x.
- The probability of the complementary event is called the error probability of A on x.

BASIC PROBABILITIES FOR RANDOMIZED ALGORITHMS

Let us consider any decision problem P.

- For any randomized algorithm A for P, and for any input x of A, let the set $S_{A,x}$ of all runs of A on x be the main sample space for the analysis of A on the input x.
- If one chooses the random variable $V_{A,x}$ that assigns 1 (0) to any run of A on x with the correct (wrong) output, then the expectation value of $V_{A,x}$ is exactly the success probability of A on x.
- The probability of the complementary event is called the error probability of A on x.
- If one takes a random variable that assigns to every computation its complexity (number of steps), then the expectation of this random variable equals the expected time complexity of A on x.

CLASSIFICATION of RANDOMIZED ALGORITHMS

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)
- A one-sided-error Monte Carlo algorithm (1MC), for a language L, is an algorithm that accepts with probability at least $\frac{1}{2}$ any input $x \in L$ and rejects for sure any input not in L;

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)
- A one-sided-error Monte Carlo algorithm (1MC), for a language L, is an algorithm that accepts with probability at least $\frac{1}{2}$ any input $x \in L$ and rejects for sure any input not in L; (The error probability of such algorithms converges to 0 with exponential speed if a number of independent runs are executed.)

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)
- A one-sided-error Monte Carlo algorithm (1MC), for a language L, is an algorithm that accepts with probability at least $\frac{1}{2}$ any input $x \in L$ and rejects for sure any input not in L; (The error probability of such algorithms converges to 0 with exponential speed if a number of independent runs are executed.)
- A bounded-error Monte Carlo algorithm (2MC) A, for a function F, is an algorithm for which there exists a constant $\varepsilon > 0$ such that, for any input x, the algorithm computes the correct output $A(x) = F(x)$, with probability at least $1/2 + \varepsilon$.

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)
- A one-sided-error Monte Carlo algorithm (1MC), for a language L, is an algorithm that accepts with probability at least $\frac{1}{2}$ any input $x \in L$ and rejects for sure any input not in L; (The error probability of such algorithms converges to 0 with exponential speed if a number of independent runs are executed.)
- A bounded-error Monte Carlo algorithm (2MC) A, for a function F, is an algorithm for which there exists a constant $\varepsilon > 0$ such that, for any input x, the algorithm computes the correct output $A(x) = F(x)$, with probability at least $1/2 + \varepsilon$. (The error probability of such an algorithm can be reduced to an arbitrarily small given constant δ using only constantly many (depending on δ) runs.)

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)
- A one-sided-error Monte Carlo algorithm (1MC), for a language L, is an algorithm that accepts with probability at least $\frac{1}{2}$ any input $x \in L$ and rejects for sure any input not in L; (The error probability of such algorithms converges to 0 with exponential speed if a number of independent runs are executed.)
- A bounded-error Monte Carlo algorithm (2MC) A, for a function F, is an algorithm for which there exists a constant $\varepsilon > 0$ such that, for any input x, the algorithm computes the correct output $A(x) = F(x)$, with probability at least $1/2 + \varepsilon$. (The error probability of such an algorithm can be reduced to an arbitrarily small given constant δ using only constantly many (depending on δ) runs.)
- A (unbounded error) Monte Carlo algorithm (UMC) is an algorithm that, for any input x, computes the correct output with probability at least $1/2$.

CLASSIFICATION of RANDOMIZED ALGORITHMS

- As Las Vegas algorithms are called those algorithms that never produce wrong outcomes, though sometimes they may produce the outcome "I don't know" (usually denoted as ??). (However, with bounded probability only.)
- A one-sided-error Monte Carlo algorithm (1MC), for a language L, is an algorithm that accepts with probability at least $\frac{1}{2}$ any input $x \in L$ and rejects for sure any input not in L; (The error probability of such algorithms converges to 0 with exponential speed if a number of independent runs are executed.)
- A bounded-error Monte Carlo algorithm (2MC) A, for a function F, is an algorithm for which there exists a constant $\varepsilon > 0$ such that, for any input x, the algorithm computes the correct output $A(x) = F(x)$, with probability at least $1/2 + \varepsilon$. (The error probability of such an algorithm can be reduced to an arbitrarily small given constant δ using only constantly many (depending on δ) runs.)
- A (unbounded error) Monte Carlo algorithm (UMC) is an algorithm that, for any input x, computes the correct output with probability at least $1/2$. (To reduce the error probability of such an algorithm below a given constant, exponentially many runs may be needed.)

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - I

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - I

Fooling the adversary (balamuting nepriatela) method: One finds for a given problem P a set of deterministic algorithms for P such that, for each input most of these algorithms compute correct results of P efficiently

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - I

Fooling the adversary (balamuting nepriatela) method: One finds for a given problem P a set of deterministic algorithms for P such that, for each input most of these algorithms compute correct results of P efficiently
and one then takes as the randomized algorithm for P a probability distribution over such a set of deterministic algorithms for P.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - I

Fooling the adversary (balamuting nepriatela) method: One finds for a given problem P a set of deterministic algorithms for P such that, for each input most of these algorithms compute correct results of P efficiently
and one then takes as the randomized algorithm for P a probability distribution over such a set of deterministic algorithms for P.

The idea is to overcome such a situation when a set S of deterministic algorithms for P is given such that for each of them there exist bad inputs (for which the algorithm gives wrong result or computes inefficiently) and for each input there exist a lot of algorithms in S giving a correct answer efficiently).

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - II

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - II

Abundance of witnesses technique. A witness y for an input x and problem P is information with which one can solve the problem P for input x more efficiently than without it.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - II

Abundance of witnesses technique. A witness y for an input x and problem P is information with which one can solve the problem P for input x more efficiently than without it.

If one finds a set S such that at least half of its elements are witnesses for P and x, then a random choice of an element in the set S leads to a witness for P and x, with probability at least $\frac{1}{2}$.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - II

Abundance of witnesses technique. A witness y for an input x and problem P is information with which one can solve the problem P for input x more efficiently than without it.

If one finds a set S such that at least half of its elements are witnesses for P and x, then a random choice of an element in the set S leads to a witness for P and x, with probability at least $\frac{1}{2}$.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - III

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - III

- Fingerprinting. For solving various problems it can be much more efficient to work with very small fingerprints (hashes) of large inputs than with such large inputs directly.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - IV

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - IV

- **Random sampling.** If there are in a set sufficiently many objects we are looking for, then a random sampling in that set can provide such an object with sufficiently high probability. This method is also called probabilistic method.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - IV

- **Random sampling.** If there are in a set sufficiently many objects we are looking for, then a random sampling in that set can provide such an object with sufficiently high probability. This method is also called probabilistic method.
- **colorred Random rounding.**: A hard to solve optimization problem P is transferred to an easy to solve optimization problem P_0, just by increasing the size of the solution space, in such a way that the outcomes of any solution of the new problem can be used to create an efficient randomized algorithm to solve the original problem P.

RANDOMIZED ALGORITHMS DESIGN PARADIGMS - IV

- **Random sampling.** If there are in a set sufficiently many objects we are looking for, then a random sampling in that set can provide such an object with sufficiently high probability. This method is also called probabilistic method.
- **colorred Random rounding.**: A hard to solve optimization problem P is transferred to an easy to solve optimization problem P_0, just by increasing the size of the solution space, in such a way that the outcomes of any solution of the new problem can be used to create an efficient randomized algorithm to solve the original problem P.
- **An amplification** of the success probability of a randomized algorithm can be achieved by repeating independent computations on the same input (but, of course, with different random auxiliary inputs).

Example for random rounding

A $\{0,1\}$-**problem** P, in which the task is to find a proper assignment of the values from the set $\{0,1\}$ to each variable,

Example for random rounding

A $\{0,1\}$-**problem** P, in which the task is to find a proper assignment of the values from the set $\{0,1\}$ to each variable, **is transferred to a** $[0,1]$-**problem** in which the task is to assign to each variable x a number $n_x \in [0,1]$ such that to solve the problem P with high probability the value 1 is assigned with the probability n_x.

STRINGS EQUALITY PROBLEM

STRINGS EQUALITY PROBLEM

Notation: For a binary vector/string $x = x_1 x_2 \dots x_n$ let

$$\text{Number}(x) = \sum_{i=1}^{n} x_i 2^{n-i}.$$

STRINGS EQUALITY PROBLEM

Notation: For a binary vector/string $x = x_1 x_2 \dots x_n$ let

$$\text{Number}(x) = \sum_{i=1}^{n} x_i 2^{n-i}.$$

Problem: Let each of the two parties, say A and B have a one n-bit string. Both parties have to decide, by communication only, whether their strings are equal. How to do that efficiently?

STRINGS EQUALITY PROBLEM

Notation: For a binary vector/string $x = x_1 x_2 \dots x_n$ let

$$\text{Number}(x) = \sum_{i=1}^{n} x_i 2^{n-i}.$$

Problem: Let each of the two parties, say A and B have a one n-bit string. Both parties have to decide, by communication only, whether their strings are equal. How to do that efficiently?

Each deterministic protocol clearly requires sending at least n bits in the worth case.

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

Initial situation.

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

Initial situation.

Party A has a binary string $x = x_1 x_2 \dots x_n$.

Party B has a binary string $y = y_1 y_2 \dots y_n$.

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

Initial situation.

Party A has a binary string $x = x_1 x_2 \dots x_n$.
Party B has a binary string $y = y_1 y_2 \dots y_n$.

Protocol:

1. Alice chooses, randomly, a prime $p \leq n^2$ and sends to Bob p and the binary representation of the number

$$s = \text{Number}(x) \bmod p;$$

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

Initial situation.

Party A has a binary string $x = x_1 x_2 \dots x_n$.
Party B has a binary string $y = y_1 y_2 \dots y_n$.

Protocol:

1. Alice chooses, randomly, a prime $p \leq n^2$ and sends to Bob p and the binary representation of the number

$$s = \text{Number}(x) \bmod p;$$

2. Bob computes

$$t = \text{Number}(y) \bmod p$$

and declares that $x = y$ iff $s = t$.

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

Initial situation.

Party A has a binary string $x = x_1 x_2 \dots x_n$.
Party B has a binary string $y = y_1 y_2 \dots y_n$.

Protocol:

1. Alice chooses, randomly, a prime $p \leq n^2$ and sends to Bob p and the binary representation of the number

$$s = \text{Number}(x) \bmod p;$$

2. Bob computes

$$t = \text{Number}(y) \bmod p$$

and declares that $x = y$ iff $s = t$.

Analysis: The protocol requires to send at most

$$2\lceil \lg n^2 \rceil \leq 4\lceil \lg n \rceil \text{ bits.}$$

STRINGS EQUALITY PROBLEM - RANDOMIZED PROTOCOL

Initial situation.

Party A has a binary string $x = x_1 x_2 \dots x_n$.
Party B has a binary string $y = y_1 y_2 \dots y_n$.

Protocol:

1. Alice chooses, randomly, a prime $p \leq n^2$ and sends to Bob p and the binary representation of the number

$$s = \text{Number}(x) \bmod p;$$

2. Bob computes

$$t = \text{Number}(y) \bmod p$$

and declares that $x = y$ iff $s = t$.

Analysis: The protocol requires to send at most

$$2\lceil \lg n^2 \rceil \leq 4\lceil \lg n \rceil \text{ bits.}$$

Example: For $n = 2^{64} \approx 10^{21}$ the protocol requires to send at most 256 bits.

ERROR ANALYSIS

Let us say that a prime $2 < p < n^2$ is bad for a pair $(x, y), x \neq y$, if the above protocol for such an input pair (x, y) and such a choice of a prime yields a wrong answer.

ERROR ANALYSIS

Let us say that a prime $2 < p < n^2$ is bad for a pair $(x, y), x \neq y$, if the above protocol for such an input pair (x, y) and such a choice of a prime yields a wrong answer.

Notation $\text{Prim}(m) =$ number of primes smaller than m.

ERROR ANALYSIS

Let us say that a prime $2 < p < n^2$ is bad for a pair $(x, y), x \neq y$, if the above protocol for such an input pair (x, y) and such a choice of a prime yields a wrong answer.

Notation $\text{Prim}(m) =$ number of primes smaller than m.

Error probability for an input (x, y) is

$$\frac{\text{number of bad primes for } (x, y)}{\text{Prim}(n^2)}.$$

ERROR ANALYSIS

Let us say that a prime $2 < p < n^2$ is bad for a pair $(x, y), x \neq y$, if the above protocol for such an input pair (x, y) and such a choice of a prime yields a wrong answer.

Notation $\text{Prim}(m) =$ number of primes smaller than m.

Error probability for an input (x, y) is

$$\frac{\text{number of bad primes for } (x, y)}{\text{Prim}(n^2)}.$$

Since, by Prime number theorem, $\text{Prim}(m) > m/\ln m$ for $m > 69$, we have that for $n \geq 9$

$$\text{Prim}(n^2) > \frac{n^2}{2 \ln n}.$$

ERROR ANALYSIS

Let us say that a prime $2 < p < n^2$ is bad for a pair $(x, y), x \neq y$, if the above protocol for such an input pair (x, y) and such a choice of a prime yields a wrong answer.

Notation $\mathrm{Prim}(m) =$ number of primes smaller than m.

Error probability for an input (x, y) is

$$\frac{\text{number of bad primes for } (x, y)}{\mathrm{Prim}(n^2)}.$$

Since, by Prime number theorem, $\mathrm{Prim}(m) > m/\ln m$ for $m > 69$, we have that for $n \geq 9$

$$\mathrm{Prim}(n^2) > \frac{n^2}{2 \ln n}.$$

The so-called Prime number theorem says that there are approximately $\frac{n}{\ln n}$ primes among the first n integers.

NUMBER Of BAD PRIMES

NUMBER Of BAD PRIMES

Lemma Number of bad primes for $x \neq y$ is at most $n - 1$.

NUMBER Of BAD PRIMES

Lemma Number of bad primes for $x \neq y$ is at most $n - 1$.

Proof. A prime p is clearly bad for the pair $(x, y), x \neq y$, if and only if p divides the number

$$w = |\text{Number}(x) - \text{Number}(y)| < 2^n$$

NUMBER Of BAD PRIMES

Lemma Number of bad primes for $x \neq y$ is at most $n-1$.

Proof. A prime p is clearly bad for the pair $(x, y), x \neq y$, if and only if p divides the number

$$w = |\text{Number}(x) - \text{Number}(y)| < 2^n$$

Observe that w can be uniquely factorized as

$$w = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$$

where $p_1 < p_2 < \ldots < p_k$ are primes.

NUMBER Of BAD PRIMES

Lemma Number of bad primes for $x \neq y$ is at most $n-1$.

Proof. A prime p is clearly bad for the pair $(x, y), x \neq y$, if and only if p divides the number

$$w = |\text{Number}(x) - \text{Number}(y)| < 2^n$$

Observe that w can be uniquely factorized as

$$w = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$$

where $p_1 < p_2 < \ldots < p_k$ are primes.

We can show that $k \leq n-1$. Indeed, if $k \geq n$, then

$$w \geq 1 \cdot 2 \cdot 3 \dots n = n! > 2^n$$

what is a contradiction.

NUMBER Of BAD PRIMES

Lemma Number of bad primes for $x \neq y$ is at most $n-1$.

Proof. A prime p is clearly bad for the pair $(x, y), x \neq y$, if and only if p divides the number

$$w = |\text{Number}(x) - \text{Number}(y)| < 2^n$$

Observe that w can be uniquely factorized as

$$w = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$$

where $p_1 < p_2 < \ldots < p_k$ are primes.

We can show that $k \leq n-1$. Indeed, if $k \geq n$, then

$$w \geq 1 \cdot 2 \cdot 3 \dots n = n! > 2^n$$

what is a contradiction.

Probability of the error of the above equality protocol is therefore:

$$\frac{n-1}{\text{Prim}(n^2)} \leq \frac{n-1}{n^2/\ln n^2} \leq \frac{\ln n^2}{n} = \frac{2 \ln n}{n}$$

which is at most $0.369 \cdot 10^{-14}$ in case $n = 10^{16}$.

PROBABILITY AMPLIFICATION

PROBABILITY AMPLIFICATION

In case the above protocol is repeated 10 times, each time with a different prime, and the answer is $x = y$ each time, then the probability of an error is at most

$$\left(\frac{\ln n^2}{n}\right)^{10}$$

PROBABILITY AMPLIFICATION

In case the above protocol is repeated 10 times, each time with a different prime, and the answer is $x = y$ each time, then the probability of an error is at most

$$\left(\frac{\ln n^2}{n}\right)^{10}$$

and therefore, for $n = 10^{16}$, the error probability is at most

$$0.47 \cdot 10^{-141}.$$

PROBABILITY AMPLIFICATION

In case the above protocol is repeated 10 times, each time with a different prime, and the answer is $x = y$ each time, then the probability of an error is at most

$$\left(\frac{\ln n^2}{n}\right)^{10}$$

and therefore, for $n = 10^{16}$, the error probability is at most

$$0.47 \cdot 10^{-141}.$$

Reminder: Probability of the correct output is at least

$$1 - \frac{2 \ln n}{n}.$$

A LAS VEGAS ALGORITHM WITH ?? - EXAMPLE

A LAS VEGAS ALGORITHM WITH ?? - EXAMPLE

Let Alice have 10 n bit strings $x_i \in \{0.1\}^n$, $1 \leq i \leq 10, n > 200$ and Bob have 10 n bit strings $y_i \in \{0,1\}^n, 1 \leq i \leq 10$. The task for them is to determine, by communication, whether there exists an i_0 such that $x_{i_0} = y_{i_0}$.

A LAS VEGAS ALGORITHM WITH ?? - EXAMPLE

Let Alice have 10 n bit strings $x_i \in \{0.1\}^n$, $1 \leq i \leq 10, n > 200$ and Bob have 10 n bit strings $y_i \in \{0,1\}^n, 1 \leq i \leq 10$. The task for them is to determine, by communication, whether there exists an i_0 such that $x_{i_0} = y_{i_0}$.

One can show that each deterministic protocol for this problem has to exchange in the worth case $10n$ bits.

A LAS VEGAS ALGORITHM WITH ?? - EXAMPLE

Let Alice have 10 n bit strings $x_i \in \{0.1\}^n$, $1 \leq i \leq 10, n > 200$ and Bob have 10 n bit strings $y_i \in \{0,1\}^n, 1 \leq i \leq 10$. The task for them is to determine, by communication, whether there exists an i_0 such that $x_{i_0} = y_{i_0}$.

One can show that each deterministic protocol for this problem has to exchange in the worth case $10n$ bits. Therefore no deterministic algorithm is essentially more efficient than sending all bits from Alice to Bob and then let Bob to make comparisons of pairs x_i and y_i until (if at all) such an i_0 is found..

A LAS VEGAS ALGORITHM WITH ?? - EXAMPLE

Let Alice have 10 n bit strings $x_i \in \{0.1\}^n$, $1 \leq i \leq 10, n > 200$ and Bob have 10 n bit strings $y_i \in \{0,1\}^n, 1 \leq i \leq 10$. The task for them is to determine, by communication, whether there exists an i_0 such that $x_{i_0} = y_{i_0}$.

One can show that each deterministic protocol for this problem has to exchange in the worth case $10n$ bits. Therefore no deterministic algorithm is essentially more efficient than sending all bits from Alice to Bob and then let Bob to make comparisons of pairs x_i and y_i until (if at all) such an i_0 is found..

We show the existence of a Las Vegas algorithm to solve the above problem (that is to decide whether such an i exists) with communication complexity

$$n + \mathcal{O}(\lg n).$$

PROTOCOL

- Alice first randomly chooses 10 primes $p_1, \ldots, p_{10}$, each smaller than n^2.

PROTOCOL

- Alice first randomly chooses 10 primes $p_1, \ldots, p_{10}$, each smaller than n^2. Afterwards, Alice computes all

$$s_i = \text{Number}(x_i) \bmod p_i, i \in \{1, .2, \ldots, 10\}$$

and, finally, she sends to Bob all 20 numbers:

$$p_1, \ldots, p_{10}, s_1, \ldots, s_{10}.$$

PROTOCOL

- Alice first randomly chooses 10 primes $p_1, \ldots, p_{10}$, each smaller than n^2. Afterwards, Alice computes all

$$s_i = \text{Number}(x_i) \bmod p_i, i \in \{1, .2, \ldots, 10\}$$

and, finally, she sends to Bob all 20 numbers:

$$p_1, \ldots, p_{10}, s_1, \ldots, s_{10}.$$

- Bob computes all numbers

$$r_i = \text{Number}(y_i) \bmod p_i$$

and compares elements in all pairs (s_i, r_i).

PROTOCOL

- Alice first randomly chooses 10 primes $p_1, \ldots, p_{10}$, each smaller than n^2. Afterwards, Alice computes all

$$s_i = \text{Number}(x_i) \bmod p_i, i \in \{1, .2, \ldots, 10\}$$

and, finally, she sends to Bob all 20 numbers:

$$p_1, \ldots, p_{10}, s_1, \ldots, s_{10}.$$

- Bob computes all numbers

$$r_i = \text{Number}(y_i) \bmod p_i$$

and compares elements in all pairs (s_i, r_i).

If $s_i \neq r_i$ for all i, then Bob outputs NO (or 0).

PROTOCOL

- Alice first randomly chooses 10 primes $p_1, \ldots, p_{10}$, each smaller than n^2. Afterwards, Alice computes all

$$s_i = \text{Number}(x_i) \bmod p_i, i \in \{1, .2, \ldots, 10\}$$

and, finally, she sends to Bob all 20 numbers:

$$p_1, \ldots, p_{10}, s_1, \ldots, s_{10}.$$

- Bob computes all numbers

$$r_i = \text{Number}(y_i) \bmod p_i$$

and compares elements in all pairs (s_i, r_i).

If $s_i \neq r_i$ for all i, then Bob outputs NO (or 0).

Otherwise, if j is the smallest integer such that $s_j = r_j$, then Bob sends to Alice the pair (j, y_j).

PROTOCOL

- Alice first randomly chooses 10 primes $p_1, \ldots, p_{10}$, each smaller than n^2. Afterwards, Alice computes all

$$s_i = \text{Number}(x_i) \bmod p_i, i \in \{1, .2, \ldots, 10\}$$

and, finally, she sends to Bob all 20 numbers:

$$p_1, \ldots, p_{10}, s_1, \ldots, s_{10}.$$

- Bob computes all numbers

$$r_i = \text{Number}(y_i) \bmod p_i$$

and compares elements in all pairs (s_i, r_i).

If $s_i \neq r_i$ for all i, then Bob outputs NO (or 0).

Otherwise, if j is the smallest integer such that $s_j = r_j$, then Bob sends to Alice the pair (j, y_j).

- If $x_j = y_j$, Alice outputs YES (or 1);
otherwise she outputs ?? (because it may exist other k with $x_k = y_k$).

COMPLEXITY ANALYSIS of the PROTOCOL 1/2

COMPLEXITY ANALYSIS of the PROTOCOL 1/2

Above protocol exchanges $20\lceil \lg n^2 \rceil + n + 4$ bits - $\{$4 bits one needs to send $j\}$.

COMPLEXITY ANALYSIS of the PROTOCOL 1/2

Above protocol exchanges $20\lceil \lg n^2 \rceil + n + 4$ bits - $\{4$ bits one needs to send $j\}$.

Proof that the protocol is a Las Vegas protocol:

COMPLEXITY ANALYSIS of the PROTOCOL 1/2

Above protocol exchanges $20\lceil \lg n^2 \rceil + n + 4$ bits - $\{4$ bits one needs to send $j\}$.

Proof that the protocol is a Las Vegas protocol:

- If $x_i \neq y_i$ for all i, then the probability that

$$\text{Number}(x_i) \bmod p_i \neq \text{Number}(y_i) \bmod p_i$$

for all i, and therefore the probability that the above protocol produces the correct output is at least

$$\left(1 - \frac{2 \ln n}{n}\right)^{10}$$

because, according to the analysis of the strings equality algorithm, the probability that for any $1 \leq i \leq 10$ $\text{Number}(x_i) \bmod p_i \neq \text{Number}(y_i) \bmod p_i$ is at least $1 - \frac{2 \ln n}{n}$.

COMPLEXITY ANALYSIS of the PROTOCOL 1/2

Above protocol exchanges $20\lceil \lg n^2 \rceil + n + 4$ bits - $\{4$ bits one needs to send $j\}$.

Proof that the protocol is a Las Vegas protocol:

- If $x_i \neq y_i$ for all i, then the probability that

$$\text{Number}(x_i) \bmod p_i \neq \text{Number}(y_i) \bmod p_i$$

for all i, and therefore the probability that the above protocol produces the correct output is at least

$$\left(1 - \frac{2 \ln n}{n}\right)^{10}$$

because, according to the analysis of the strings equality algorithm, the probability that for any $1 \leq i \leq 10$ $\text{Number}(x_i) \bmod p_i \neq \text{Number}(y_i) \bmod p_i$ is at least $1 - \frac{2 \ln n}{n}$.

Moreover, it can be shown that for for sufficiently large n.

$$\left(1 - \frac{2 \ln n}{n}\right)^{10} \geq 1 - \frac{20 \ln n}{n} \geq \frac{1}{2}.$$

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

  Protocol then accepts the input iff

  $$\text{Number}(x_i) \bmod p_i \neq \text{Number}(x_i) \bmod p_i$$

  for all $i < j_0$. Let us denote such an event by E_{j_0}.

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

  Protocol then accepts the input iff

  $$\text{Number}(x_i) \bmod p_i \neq \text{Number}(x_i) \bmod p_i$$

  for all $i < j_0$. Let us denote such an event by E_{j_0}.

  If $j_0 = 1$, then the protocol accepts the input with certainty.

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

  Protocol then accepts the input iff

  $$\text{Number}(x_i) \bmod p_i \neq \text{Number}(x_i) \bmod p_i$$

  for all $i < j_0$. Let us denote such an event by E_{j_0}.

  If $j_0 = 1$, then the protocol accepts the input with certainty. If $j_0 > 1$, then, as discussed before, probability of the event E_{j_0} is at least

  $$\left(1 - \frac{2 \ln n}{n}\right)^{j_0 - 1} \geq 1 - \frac{2(j_0 - 1) \ln n}{n}$$

  for sufficiently large n

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

  Protocol then accepts the input iff

  $$\text{Number}(x_i) \bmod p_i \neq \text{Number}(x_i) \bmod p_i$$

  for all $i < j_0$. Let us denote such an event by E_{j_0}.

  If $j_0 = 1$, then the protocol accepts the input with certainty. If $j_0 > 1$, then, as discussed before, probability of the event E_{j_0} is at least

  $$\left(1 - \frac{2 \ln n}{n}\right)^{j_0 - 1} \geq 1 - \frac{2(j_0 - 1) \ln n}{n}$$

  for sufficiently large n

  and therefore the protocol outputs YES (1) with probability at least $1 - \frac{18 \ln n}{n}$, which is larger than $\frac{1}{2}$ for all $n \geq 189$.

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

  Protocol then accepts the input iff

  $$\text{Number}(x_i) \bmod p_i \neq \text{Number}(x_i) \bmod p_i$$

  for all $i < j_0$. Let us denote such an event by E_{j_0}.

  If $j_0 = 1$, then the protocol accepts the input with certainty. If $j_0 > 1$, then, as discussed before, probability of the event E_{j_0} is at least

  $$\left(1 - \frac{2 \ln n}{n}\right)^{j_0 - 1} \geq 1 - \frac{2(j_0 - 1) \ln n}{n}$$

  for sufficiently large n

  and therefore the protocol outputs YES (1) with probability at least $1 - \frac{18 \ln n}{n}$, which is larger than $\frac{1}{2}$ for all $n \geq 189$.

  In the again complementary case, when there is an $l < j_0$ such that

  $$\text{Number}(x_l) \bmod p_l = \text{Number}(x_l) \bmod p_l$$

  the protocol produces as output ??

COMPLEXITY ANALYSIS of the PROTOCOL 2/2

- Let us now consider the complementary case - namely that there is a j such that $x_j = y_j$ and let j_0 be the smallest such j.

  Protocol then accepts the input iff

  $$\text{Number}(x_i) \bmod p_i \neq \text{Number}(x_i) \bmod p_i$$

  for all $i < j_0$. Let us denote such an event by E_{j_0}.

  If $j_0 = 1$, then the protocol accepts the input with certainty. If $j_0 > 1$, then, as discussed before, probability of the event E_{j_0} is at least

  $$\left(1 - \frac{2 \ln n}{n}\right)^{j_0 - 1} \geq 1 - \frac{2(j_0 - 1) \ln n}{n}$$

  for sufficiently large n

  and therefore the protocol outputs YES (1) with probability at least $1 - \frac{18 \ln n}{n}$, which is larger than $\frac{1}{2}$ for all $n \geq 189$.

  In the again complementary case, when there is an $l < j_0$ such that

  $$\text{Number}(x_l) \bmod p_l = \text{Number}(x_l) \bmod p_l$$

  the protocol produces as output ?? and is indeed Las vegaas protocol.

TWO TYPES of LAS VEGAS ALGORITHMS

TWO TYPES of LAS VEGAS ALGORITHMS

As already defines, there are two types of Las Vegas algorithms:

- Algorithms that never produce ??.

TWO TYPES of LAS VEGAS ALGORITHMS

As already defines, there are two types of Las Vegas algorithms:

- Algorithms that never produce ??.
- Algorithms that may produce ??.

TWO TYPES of LAS VEGAS ALGORITHMS

As already defines, there are two types of Las Vegas algorithms:

- Algorithms that never produce ??.
- Algorithms that may produce ??.

Note: Las Vegas algorithms may not terminate for some inputs!!!

TWO TYPES of LAS VEGAS ALGORITHMS

As already defines, there are two types of Las Vegas algorithms:

- Algorithms that never produce ??.
- Algorithms that may produce ??.

Note: Las Vegas algorithms may not terminate for some inputs!!!

Claim: Any Las Vegas algorithm A_1 can be converted to a Las Vegas algorithm A_2 that solves the same problem and never produces ??.

TWO TYPES of LAS VEGAS ALGORITHMS

As already defines, there are two types of Las Vegas algorithms:

- Algorithms that never produce ??.
- Algorithms that may produce ??.

Note: Las Vegas algorithms may not terminate for some inputs!!!

Claim: Any Las Vegas algorithm A_1 can be converted to a Las Vegas algorithm A_2 that solves the same problem and never produces ??.

Construction of A_2 is simple. Each time A_1 is to produce the output ??, what can be done only with bounded probability, a new run of A_1 is initialized with the same input.

AMPLIFICATION of 1MC ALGORITHMS

AMPLIFICATION of 1MC ALGORITHMS

Error probability of 1MC algorithms decreases exponentially with the number of repetitions of computations.

AMPLIFICATION of 1MC ALGORITHMS

Error probability of 1MC algorithms decreases exponentially with the number of repetitions of computations.

Indeed, if we have k independent runs of the algorithm and one output is 1 (accepted), then the input is accepted with certainty.

AMPLIFICATION of 1MC ALGORITHMS

Error probability of 1MC algorithms decreases exponentially with the number of repetitions of computations.

Indeed, if we have k independent runs of the algorithm and one output is 1 (accepted), then the input is accepted with certainty.

If all outputs are 0 (rejection), then the output will be NO (rejection) and the error probability is at most $\left(\frac{1}{2}\right)^k$.

AMPLIFICATION of 1MC ALGORITHMS

Error probability of 1MC algorithms decreases exponentially with the number of repetitions of computations.

Indeed, if we have k independent runs of the algorithm and one output is 1 (accepted), then the input is accepted with certainty.

If all outputs are 0 (rejection), then the output will be NO (rejection) and the error probability is at most $\left(\frac{1}{2}\right)^k$.

Because of the exponential decrease of error probability using repeated applications, 1MC algorithms are very popular.

AMPLIFICATION of 2MC ALGORITHMS

AMPLIFICATION of 2MC ALGORITHMS

Let A be a 2MC algorithm for a function F and $\varepsilon > 0$ such that

$$\text{Prob}(A(x) = F(x)) \geq \frac{1}{2} + \varepsilon.$$

AMPLIFICATION of 2MC ALGORITHMS

Let A be a 2MC algorithm for a function F and $\varepsilon > 0$ such that

$$\text{Prob}(A(x) = F(x)) \geq \frac{1}{2} + \varepsilon.$$

For any integer k let A_k be the algorithm that performs k independent runs of A and if there is an α that appears at least $\lceil \frac{k}{2} \rceil$ times as the output, then A_k produces α as the output;

AMPLIFICATION of 2MC ALGORITHMS

Let A be a 2MC algorithm for a function F and $\varepsilon > 0$ such that

$$\text{Prob}(A(x) = F(x)) \geq \frac{1}{2} + \varepsilon.$$

For any integer k let A_k be the algorithm that performs k independent runs of A and if there is an α that appears at least $\lceil \frac{k}{2} \rceil$ times as the output, then A_k produces α as the output; if there is no such an α, A_k produces ?? as the output.

One can show that if an $\delta > 0$ is fixed, then

$$\text{Prob}(A_k(x) = F(x)) \geq 1 - \delta$$

if

$$k \geq \frac{2 \ln \delta}{\ln(1 - 4\varepsilon^2)}.$$

AMPLIFICATION of 2MC ALGORITHMS

Let A be a 2MC algorithm for a function F and $\varepsilon > 0$ such that

$$\text{Prob}(A(x) = F(x)) \geq \frac{1}{2} + \varepsilon.$$

For any integer k let A_k be the algorithm that performs k independent runs of A and if there is an α that appears at least $\lceil \frac{k}{2} \rceil$ times as the output, then A_k produces α as the output; if there is no such an α, A_k produces ?? as the output.

One can show that if an $\delta > 0$ is fixed, then

$$\text{Prob}(A_k(x) = F(x)) \geq 1 - \delta$$

if

$$k \geq \frac{2 \ln \delta}{\ln(1 - 4\varepsilon^2)}.$$

If ε and δ are considered as constant, then so is k and therefore

$$\text{Time}_{A_k}(n) = \mathcal{O}(\text{Time}_A(n)).$$

2MC versus UMC algorithms

2MC versus UMC algorithms

Basic question: What is the difference between 2MC and UMC algorithms?

2MC versus UMC algorithms

Basic question: What is the difference between 2MC and UMC algorithms?

Answer: For an UMC algorithm A it may happen that the distance between the error probability and $\frac{1}{2}$ tends to 0 with growing input size.

2MC versus UMC algorithms

Basic question: What is the difference between 2MC and UMC algorithms?

Answer: For an UMC algorithm A it may happen that the distance between the error probability and $\frac{1}{2}$ tends to 0 with growing input size.

As a consequence, if we design, given an $\delta > 0$, an algorithm A_k, that performs k independent runs of A and

$$\text{Prob}(A_k(x)) = F(x)) > 1 - \delta$$

then running time of A_k may be exponential in the input length.

SECRET SHARING between TWO

SECRET SHARING between TWO

Problem: The task is to "partition", by a moderator, a given binary-string secret S between two parties P_1 and P_2 in such a way that none of the parties alone has the slightest idea what S is, but if they get together they can easily determine S.

SECRET SHARING between TWO

Problem: The task is to "partition", by a moderator, a given binary-string secret S between two parties P_1 and P_2 in such a way that none of the parties alone has the slightest idea what S is, but if they get together they can easily determine S.

Method: A moderator distributes a binary-string secret S, between two parties P_1 and P_2 by choosing a random binary string b, of the same length as S, and sends:

SECRET SHARING between TWO

Problem: The task is to "partition", by a moderator, a given binary-string secret S between two parties P_1 and P_2 in such a way that none of the parties alone has the slightest idea what S is, but if they get together they can easily determine S.

Method: A moderator distributes a binary-string secret S, between two parties P_1 and P_2 by choosing a random binary string b, of the same length as S, and sends:

- b to P_1 and

SECRET SHARING between TWO

Problem: The task is to "partition", by a moderator, a given binary-string secret S between two parties P_1 and P_2 in such a way that none of the parties alone has the slightest idea what S is, but if they get together they can easily determine S.

Method: A moderator distributes a binary-string secret S, between two parties P_1 and P_2 by choosing a random binary string b, of the same length as S, and sends:

- b to P_1 and
- $S \oplus b$ to P_2.

SECRET SHARING between TWO

Problem: The task is to "partition", by a moderator, a given binary-string secret S between two parties P_1 and P_2 in such a way that none of the parties alone has the slightest idea what S is, but if they get together they can easily determine S.

Method: A moderator distributes a binary-string secret S, between two parties P_1 and P_2 by choosing a random binary string b, of the same length as S, and sends:

- b to P_1 and
- $S \oplus b$ to P_2.

This way, none of the parties P_1 and P_2 alone has a slightest idea about S,

SECRET SHARING between TWO

Problem: The task is to "partition", by a moderator, a given binary-string secret S between two parties P_1 and P_2 in such a way that none of the parties alone has the slightest idea what S is, but if they get together they can easily determine S.

Method: A moderator distributes a binary-string secret S, between two parties P_1 and P_2 by choosing a random binary string b, of the same length as S, and sends:

- b to P_1 and
- $S \oplus b$ to P_2.

This way, none of the parties P_1 and P_2 alone has a slightest idea about S, but both together easily recover S by computing

$$b \oplus (S \oplus b) = S.$$

PERFECT MATCHING ALGORITHM - I.

Let $G = \langle V, E \rangle$ be an undirected graph. A subset $X \subseteq E$ is said to be a *matching* of G if no two edges in X have a common node.

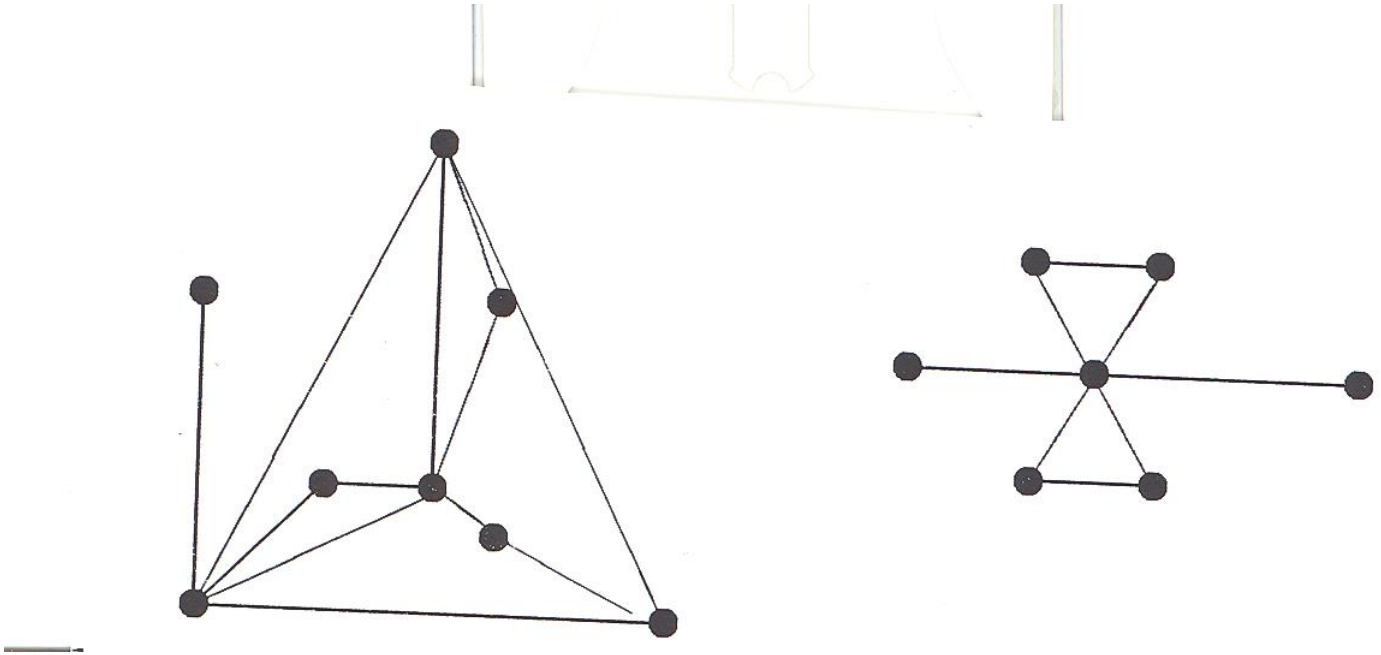
PERFECT MATCHING ALGORITHM - I.

Let $G = \langle V, E \rangle$ be an undirected graph. A subset $X \subseteq E$ is said to be a *matching* of G if no two edges in X have a common node.
A matching is said to be a *perfect matching* if its edges **cover** all nodes of G.

PERFECT MATCHING ALGORITHM - I.

Let $G = \langle V, E \rangle$ be an undirected graph. A subset $X \subseteq E$ is said to be a *matching* of G if no two edges in X have a common node.
A matching is said to be a *perfect matching* if its edges **cover** all nodes of G.

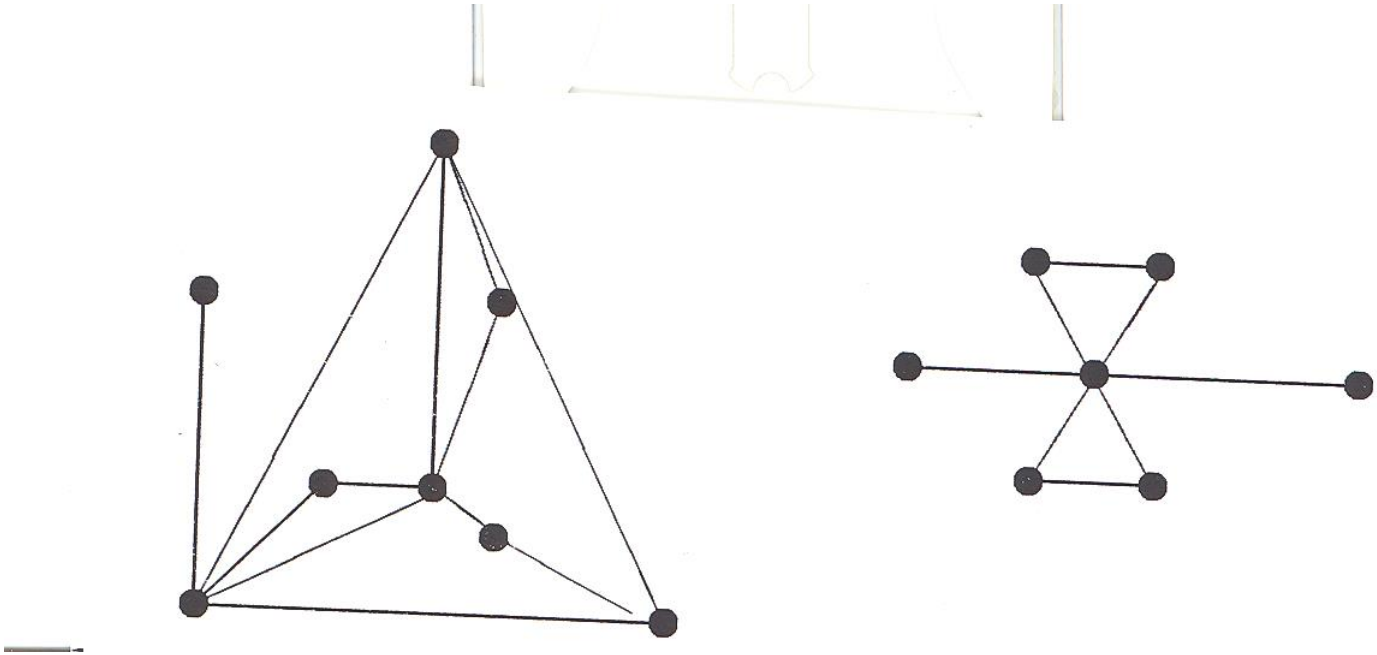
Example: Which of the graphs in the next figure has a perfect matching?



PERFECT MATCHING ALGORITHM - I.

Let $G = \langle V, E \rangle$ be an undirected graph. A subset $X \subseteq E$ is said to be a *matching* of G if no two edges in X have a common node.
A matching is said to be a *perfect matching* if its edges **cover** all nodes of G.

Example: Which of the graphs in the next figure has a perfect matching?



There are polynomial time algorithms to decide whether a given graph has perfect matching, but none is so simple as the randomized algorithm based on so called Tutte theorem presented bellow.

TUTTE MATRIX

TUTTE MATRIX

Basic concept: Tutte matrix of a graph

Let $G = \langle V, E \rangle$ be a graph with nodes $V = \{1, 2, \ldots, n\}$, $|V|$ even. The *Tutte matrix* $A_G = \{a_{ij}\}_{i,j=1}^{n}$ of G is defined by

$$a_{ij} = \left\{ \begin{array}{cl} x_{ij} & \text{if } (i,j) \in E,\ i < j; \\ -x_{ij} & \text{if } (i,j) \in E,\ i > j; \\ 0 & \text{if } (i,j) \notin E \end{array} \right\},$$

where x_{ij} are variables, all different for different i, j.

TUTTE THEOREM - I.

Example: For the graph of six nodes at which node 1 is connected with nodes 4 and 5, node 2 with nodes 5 and 6 and the node 3 is connected with nodes 5 and 6 the Tutte matrix has the form:

TUTTE THEOREM - I.

Example: For the graph of six nodes at which node 1 is connected with nodes 4 and 5, node 2 with nodes 5 and 6 and the node 3 is connected with nodes 5 and 6 the Tutte matrix has the form:

$$\begin{pmatrix} 0 & 0 & 0 & x_{14} & x_{15} & 0 \\ 0 & 0 & 0 & 0 & x_{25} & x_{26} \\ 0 & 0 & 0 & 0 & x_{35} & x_{36} \\ -x_{14} & 0 & 0 & 0 & 0 & 0 \\ -x_{15} & -x_{25} & -x_{35} & 0 & 0 & 0 \\ 0 & -x_{26} & -x_{36} & 0 & 0 & 0 \end{pmatrix}$$

TUTTE THEOREM - I.

Example: For the graph of six nodes at which node 1 is connected with nodes 4 and 5, node 2 with nodes 5 and 6 and the node 3 is connected with nodes 5 and 6 the Tutte matrix has the form:

$$\begin{pmatrix} 0 & 0 & 0 & x_{14} & x_{15} & 0 \\ 0 & 0 & 0 & 0 & x_{25} & x_{26} \\ 0 & 0 & 0 & 0 & x_{35} & x_{36} \\ -x_{14} & 0 & 0 & 0 & 0 & 0 \\ -x_{15} & -x_{25} & -x_{35} & 0 & 0 & 0 \\ 0 & -x_{26} & -x_{36} & 0 & 0 & 0 \end{pmatrix}$$

Tutte theorem: A graph $G = \langle V, E \rangle$, with $|V|$ even, has a perfect matching iff the determinant of the corresponding Tutte matrix is not identically zero.

TUTTE THEOREM - I.

Example: For the graph of six nodes at which node 1 is connected with nodes 4 and 5, node 2 with nodes 5 and 6 and the node 3 is connected with nodes 5 and 6 the Tutte matrix has the form:

$$\begin{pmatrix} 0 & 0 & 0 & x_{14} & x_{15} & 0 \\ 0 & 0 & 0 & 0 & x_{25} & x_{26} \\ 0 & 0 & 0 & 0 & x_{35} & x_{36} \\ -x_{14} & 0 & 0 & 0 & 0 & 0 \\ -x_{15} & -x_{25} & -x_{35} & 0 & 0 & 0 \\ 0 & -x_{26} & -x_{36} & 0 & 0 & 0 \end{pmatrix}$$

Tutte theorem: A graph $G = \langle V, E \rangle$, with $|V|$ even, has a perfect matching iff the determinant of the corresponding Tutte matrix is not identically zero.

Proof: The determinant of A_G equals $\sum_\pi \sigma_\pi \prod_{i=1}^{n} a_{i\pi(i)}$ where π are permutations of $\{1, 2, \ldots, n\}$ and $\sigma_\pi = 1$ ($\sigma_\pi = -1$) if π is a product of an even (odd) number of transpositions.

TUTTE THEOREM - OBSERVATIONS

TUTTE THEOREM - OBSERVATIONS

Observation 1: For a permutation π, $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ iff $G_\pi = \{(i, \pi(i)),\ 1 \leq i \leq n\}$ is a subgraph of G.

TUTTE THEOREM - OBSERVATIONS

Observation 1: For a permutation π, $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ iff $G_\pi = \{(i, \pi(i)),\ 1 \leq i \leq n\}$ is a subgraph of G.

Observation 2: Permutations π with at least one odd cycle do not contribute at all to the determinant of A, because to each such permutation π there is a permutation π' such that $\prod_{i=1}^{n} a_{i\pi(i)} = -\prod_{i=1}^{n} a_{i\pi'(i)}$

TUTTE THEOREM - OBSERVATIONS

Observation 1: For a permutation π, $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ iff $G_\pi = \{(i, \pi(i)),\ 1 \leq i \leq n\}$ is a subgraph of G.

Observation 2: Permutations π with at least one odd cycle do not contribute at all to the determinant of A, because to each such permutation π there is a permutation π' such that $\prod_{i=1}^{n} a_{i\pi(i)} = -\prod_{i=1}^{n} a_{i\pi'(i)}$

Observation 3: It is sufficient to consider permutations π such that G_π consists only of even cycles.

TUTTE THEOREM - OBSERVATIONS

Observation 1: For a permutation π, $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ iff $G_\pi = \{(i, \pi(i)),\ 1 \leq i \leq n\}$ is a subgraph of G.

Observation 2: Permutations π with at least one odd cycle do not contribute at all to the determinant of A, because to each such permutation π there is a permutation π' such that $\prod_{i=1}^{n} a_{i\pi(i)} = -\prod_{i=1}^{n} a_{i\pi'(i)}$

Observation 3: It is sufficient to consider permutations π such that G_π consists only of even cycles.

Notation: Let permutation π^r be obtained from π by reversing all cycles.

TUTTE THEOREM - OBSERVATIONS

Observation 1: For a permutation π, $\prod_{i=1}^{n} a_{i\pi(i)} \neq 0$ iff $G_\pi = \{(i, \pi(i)),\ 1 \leq i \leq n\}$ is a subgraph of G.

Observation 2: Permutations π with at least one odd cycle do not contribute at all to the determinant of A, because to each such permutation π there is a permutation π' such that $\prod_{i=1}^{n} a_{i\pi(i)} = -\prod_{i=1}^{n} a_{i\pi'(i)}$

Observation 3: It is sufficient to consider permutations π such that G_π consists only of even cycles.

Notation: Let permutation π^r be obtained from π by reversing all cycles.

Notation For a perfect matching E' let $t_{E'}$ denote the product of the $a's$ corresponding to the edges of E'.

TUTTE THEOREM - CASE ANALYSIS

TUTTE THEOREM - CASE ANALYSIS

Case I: $\pi = \pi^r \Rightarrow G_\pi$ consist of the cycles of length 2, π corresponds to a perfect matching E' such that $\prod_{i=1}^{n} a_{i\pi(i)} = (t_{E'})^2$.

TUTTE THEOREM - CASE ANALYSIS

Case I: $\pi = \pi^r \Rightarrow G_\pi$ consist of the cycles of length 2, π corresponds to a perfect matching E' such that $\prod_{i=1}^{n} a_{i\pi(i)} = (t_{E'})^2$.

Case II: $\pi \neq \pi^r$ In this case both π and π^r correspond to the union of two perfect matchings E, and E' obtained by alternatively selecting edges within the cycles so that

$$\prod_{i=1}^{n} a_{i\pi(i)} + \prod_{i=1}^{n} a_{i\pi^r(i)} = 2t_E t_{E'}.$$

TUTTE THEOREM - CASE ANALYSIS

Case I: $\pi = \pi^r \Rightarrow G_\pi$ consist of the cycles of length 2, π corresponds to a perfect matching E' such that $\prod_{i=1}^{n} a_{i\pi(i)} = (t_{E'})^2$.

Case II: $\pi \neq \pi^r$ In this case both π and π^r correspond to the union of two perfect matchings E, and E' obtained by alternatively selecting edges within the cycles so that

$$\prod_{i=1}^{n} a_{i\pi(i)} + \prod_{i=1}^{n} a_{i\pi^r(i)} = 2t_E t_{E'}.$$

Conclusion;

$$det(A_G) = (t_{E'_1} + \cdots + t_{E'_k})^2$$

where E'_i denotes i-th perfect matching.

EXPLANATION

EXPLANATION

Let us illustrate claims from the previous slide for a graph with 4 nodes: 1, 2, 3 ,4

EXPLANATION

Let us illustrate claims from the previous slide for a graph with 4 nodes: 1, 2, 3 ,4

Case 1. Let us have edges 1-2, 3-4 only and permutation

$$\pi = \pi^r : 1 \to 2, 2 \to 1, 3 \to 4, 4 \to 3$$

Perfect matching E' is $1-2, 3-4$, $t_{E'} = a_{12}a_{34}$ and

$$\prod_{i=1}^{4} a_{i\pi(i)} = a_{12}a_{21}a_{34}a_{43} = x_{12}x_{12}x_{34}x_{34} = (t_{E'})^2$$

EXPLANATION

Let us illustrate claims from the previous slide for a graph with 4 nodes: 1, 2, 3 ,4

Case 1. Let us have edges 1-2, 3-4 only and permutation

$$\pi = \pi^r : 1 \to 2, 2 \to 1, 3 \to 4, 4 \to 3$$

Perfect matching E' is $1-2, 3-4$, $t_{E'} = a_{12}a_{34}$ and

$$\prod_{i=1}^{4} a_{i\pi(i)} = a_{12}a_{21}a_{34}a_{43} = x_{12}x_{12}x_{34}x_{34} = (t_{E'})^2$$

Case 2. Let us have edges 1-2, 2-3, 3-4, 4-1 and permutations:

$$\pi : 1 \to 2 \to 3 \to 4 \to 1, \quad \pi^r : 1 \to 4 \to 3 \to 2 \to 1$$

EXPLANATION

Let us illustrate claims from the previous slide for a graph with 4 nodes: 1, 2, 3 ,4

Case 1. Let us have edges 1-2, 3-4 only and permutation

$$\pi = \pi^r : 1 \to 2, 2 \to 1, 3 \to 4, 4 \to 3$$

Perfect matching E' is $1-2, 3-4$, $t_{E'} = a_{12}a_{34}$ and

$$\prod_{i=1}^{4} a_{i\pi(i)} = a_{12}a_{21}a_{34}a_{43} = x_{12}x_{12}x_{34}x_{34} = (t_{E'})^2$$

Case 2. Let us have edges 1-2, 2-3, 3-4, 4-1 and permutations:

$$\pi : 1 \to 2 \to 3 \to 4 \to 1, \quad \pi^r : 1 \to 4 \to 3 \to 2 \to 1$$

$$t_E = a_{12}a_{34}, t_{E'} = a_{41}a_{23}$$

EXPLANATION

Let us illustrate claims from the previous slide for a graph with 4 nodes: 1, 2, 3 ,4

Case 1. Let us have edges 1-2, 3-4 only and permutation

$$\pi = \pi^r : 1 \to 2, 2 \to 1, 3 \to 4, 4 \to 3$$

Perfect matching E' is $1-2, 3-4$, $t_{E'} = a_{12}a_{34}$ and

$$\prod_{i=1}^{4} a_{i\pi(i)} = a_{12}a_{21}a_{34}a_{43} = x_{12}x_{12}x_{34}x_{34} = (t_{E'})^2$$

Case 2. Let us have edges 1-2, 2-3, 3-4, 4-1 and permutations:

$$\pi : 1 \to 2 \to 3 \to 4 \to 1, \quad \pi^r : 1 \to 4 \to 3 \to 2 \to 1$$

$$t_E = a_{12}a_{34}, t_{E'} = a_{41}a_{23}$$

$$\prod_{i=1}^{4} a_{i\pi(i)} + \prod_{i=1}^{4} a_{i\pi^r(i)} = a_{12}a_{23}a_{34}a_{41} + a_{12}a_{23}a_{34}a_{41} = 2t_E t_{E'}.$$

RANDOM SELECT

Problem Given is a set $S = \{a_1, \ldots, a_n\}$ of $n > 0$ different numbers, and $1 \leq k \leq n$, find the k-th smallest number from S.

RANDOM SELECT

Problem Given is a set $S = \{a_1, \ldots, a_n\}$ of $n > 0$ different numbers, and $1 \leq k \leq n$, find the k-th smallest number from S.

A naive way to solve the problem is to sort at first S. This requires $\mathcal{O}(n \lg n)$ comparisons. The following randomized algorithm RSELECT can do that in $\mathcal{O}(n)$ steps:

RANDOM SELECT

Problem Given is a set $S = \{a_1, \ldots, a_n\}$ of $n > 0$ different numbers, and $1 \leq k \leq n$, find the k-th smallest number from S.

A naive way to solve the problem is to sort at first S. This requires $\mathcal{O}(n \lg n)$ comparisons. The following randomized algorithm RSELECT can do that in $\mathcal{O}(n)$ steps:

Algorithm RSELECT(S, k))

1. If $n = 1$ output a_1.

RANDOM SELECT

Problem Given is a set $S = \{a_1, \ldots, a_n\}$ of $n > 0$ different numbers, and $1 \leq k \leq n$, find the k-th smallest number from S.

A naive way to solve the problem is to sort at first S. This requires $\mathcal{O}(n \lg n)$ comparisons. The following randomized algorithm RSELECT can do that in $\mathcal{O}(n)$ steps:

Algorithm RSELECT(S, k))

1. If $n = 1$ output a_1.
2. Otherwise choose $i \in_r \{1, 2, \ldots, n\}$ and
    1. compute

RANDOM SELECT

Problem Given is a set $S = \{a_1, \ldots, a_n\}$ of $n > 0$ different numbers, and $1 \leq k \leq n$, find the k-th smallest number from S.

A naive way to solve the problem is to sort at first S. This requires $\mathcal{O}(n \lg n)$ comparisons. The following randomized algorithm RSELECT can do that in $\mathcal{O}(n)$ steps:

Algorithm RSELECT(S, k))

1. If $n = 1$ output a_1.
2. Otherwise choose $i \in_r \{1, 2, \ldots, n\}$ and
   1. compute
   $$S_< = \{b \in S \mid b < a_i\}$$
   $$S_> = \{b \in S \mid b > a_i\}$$
   2. **if** $|S_<| \geq k$ **then** RSELECT$(S_<, k)$;
   else if $|S_<| = k - 1$ **then** output a_i; **else**
   RSELECT$(S_>, k - |S_<| - 1)$

RANDOM SELECT

Problem Given is a set $S = \{a_1, \ldots, a_n\}$ of $n > 0$ different numbers, and $1 \leq k \leq n$, find the k-th smallest number from S.

A naive way to solve the problem is to sort at first S. This requires $\mathcal{O}(n \lg n)$ comparisons. The following randomized algorithm RSELECT can do that in $\mathcal{O}(n)$ steps:

Algorithm RSELECT(S, k))

1. If $n = 1$ output a_1.
2. Otherwise choose $i \in_r \{1, 2, \ldots, n\}$ and
   1. compute
   $$S_< = \{b \in S \mid b < a_i\}$$
   $$S_> = \{b \in S \mid b > a_i\}$$
   2. **if** $|S_<| \geq k$ **then** RSELECT$(S_<, k)$;
   else if $|S_<| = k - 1$ **then** output a_i; **else**
   RSELECT$(S_>, k - |S_<| - 1)$

This is clearly a Las Vegas algorithm.

CLASSIFICATION of RANDOMIZED OPTIMIZATION ALGORITHMS

CLASSIFICATION of RANDOMIZED OPTIMIZATION ALGORITHMS

Classification of the randomized algorithms we had so far was based on the frequency of correct outputs and it is suited only for classification of algorithms for decision problems and for computation of functions, but not for optimization problems.

CLASSIFICATION of RANDOMIZED OPTIMIZATION ALGORITHMS

Classification of the randomized algorithms we had so far was based on the frequency of correct outputs and it is suited only for classification of algorithms for decision problems and for computation of functions, but not for optimization problems.

In case of optimization problems, we do not take as the output the most frequent output from several runs, but the best output according to some optimization criterion.

CLASSIFICATION of RANDOMIZED OPTIMIZATION ALGORITHMS

Classification of the randomized algorithms we had so far was based on the frequency of correct outputs and it is suited only for classification of algorithms for decision problems and for computation of functions, but not for optimization problems.

In case of optimization problems, we do not take as the output the most frequent output from several runs, but the best output according to some optimization criterion.

Moreover, in case of optimization problems our goal is not always to find an optimal solution. We are usually quite happy to find (and in a feasible way) an almost optimal solution - whose cost (quality) does not differ much from the cost (quality) of some optimal solution.

CLASSIFICATION of RANDOMIZED OPTIMIZATION ALGORITHMS

Classification of the randomized algorithms we had so far was based on the frequency of correct outputs and it is suited only for classification of algorithms for decision problems and for computation of functions, but not for optimization problems.

In case of optimization problems, we do not take as the output the most frequent output from several runs, but the best output according to some optimization criterion.

Moreover, in case of optimization problems our goal is not always to find an optimal solution. We are usually quite happy to find (and in a feasible way) an almost optimal solution - whose cost (quality) does not differ much from the cost (quality) of some optimal solution.

All that means that a different approach to the classification of randomized approximation algorithms is needed.

ILLUSTRATION

If a randomized algorithm A computes an optimal solution for an input x only with probability at

ILLUSTRATION

If a randomized algorithm A computes an optimal solution for an input x only with probability at least $\frac{1}{|x|}$, then it does not mean that A is not useful.

ILLUSTRATION

If a randomized algorithm A computes an optimal solution for an input x only with probability at least $\frac{1}{|x|}$, then it does not mean that A is not useful.

Indeed, one can execute $|x|$ independent runs of A on given input x, call that to be the A_x algorithm, and then take the best output of all of them.

ILLUSTRATION

If a randomized algorithm A computes an optimal solution for an input x only with probability at least $\frac{1}{|x|}$, then it does not mean that A is not useful.

Indeed, one can execute $|x|$ independent runs of A on given input x, call that to be the A_x algorithm, and then take the best output of all of them. Let us explore now what is the probability of success of such an A_x algorithm.

Probability of not computing an optimal solution in one run is at most

$$1 - \frac{1}{|x|}$$

and therefore the probability that A_x does not find an optimal solution in $|x|$ independent runs is at most

$$(1 - \frac{1}{|x|})^{|x|} < \frac{1}{e}$$

ILLUSTRATION

If a randomized algorithm A computes an optimal solution for an input x only with probability at least $\frac{1}{|x|}$, then it does not mean that A is not useful.

Indeed, one can execute $|x|$ independent runs of A on given input x, call that to be the A_x algorithm, and then take the best output of all of them. Let us explore now what is the probability of success of such an A_x algorithm.

Probability of not computing an optimal solution in one run is at most

$$1 - \frac{1}{|x|}$$

and therefore the probability that A_x does not find an optimal solution in $|x|$ independent runs is at most

$$(1 - \frac{1}{|x|})^{|x|} < \frac{1}{e}$$

and we have a constant probability $1 - \frac{1}{e}$ of computing an optimal solution.

DEFINITION of OPTIMIZATION PROBLEMS

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

1. Σ_I is an input alphabet;

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

1. Σ_I is an input alphabet;
2. Σ_O is an output alphabet;

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

1. Σ_I is an input alphabet;
2. Σ_O is an output alphabet;
3. $L \subseteq \Sigma_I^*$ is the language of feasible inputs and any $x \in L$ is called a problem instance of P;

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

1. Σ_I is an input alphabet;
2. Σ_O is an output alphabet;
3. $L \subseteq \Sigma_I^*$ is the language of feasible inputs and any $x \in L$ is called a problem instance of P;
4. $\mathcal{M}$ is a function from L to $2^{\Sigma_O^*}$, and for each $x \in L$, $\mathcal{M}(x)$ is the set of feasible solutions for x;

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

1. Σ_I is an input alphabet;
2. Σ_O is an output alphabet;
3. $L \subseteq \Sigma_I^*$ is the language of feasible inputs and any $x \in L$ is called a problem instance of P;
4. $\mathcal{M}$ is a function from L to $2^{\Sigma_O^*}$, and for each $x \in L$, $\mathcal{M}(x)$ is the set of feasible solutions for x;
5. cost is a function: $\bigcup_{x \in L}(\mathcal{M}(x) \times x) \to \mathbf{R}^+$, called the cost function;

DEFINITION of OPTIMIZATION PROBLEMS

Definition: An optimization problem is a 6-tuple $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, where

1. Σ_I is an input alphabet;
2. Σ_O is an output alphabet;
3. $L \subseteq \Sigma_I^*$ is the language of feasible inputs and any $x \in L$ is called a problem instance of P;
4. $\mathcal{M}$ is a function from L to $2^{\Sigma_O^*}$, and for each $x \in L$, $\mathcal{M}(x)$ is the set of feasible solutions for x;
5. cost is a function: $\bigcup_{x \in L}(\mathcal{M}(x) \times x) \rightarrow \mathbf{R}^+$, called the cost function;
6. goal $\in$ (minimum, maximum) is an objective.

SOLVING an OPTIMIZATION PROBLEM

SOLVING an OPTIMIZATION PROBLEM

Observe that to solve an optimization problem is more as computing a relation, than as computing a function, because we are usually happy with finding one of the optimal solutions or even with a solution quite close to an optimal one.

SOLVING an OPTIMIZATION PROBLEM

Observe that to solve an optimization problem is more as computing a relation, than as computing a function, because we are usually happy with finding one of the optimal solutions or even with a solution quite close to an optimal one.

A feasible solution $\alpha \in \mathcal{M}(x)$ is called optimal for the problem instance x of $\mathcal{P}$ if

$$\text{cost}(\alpha, x) = \text{goal}\{\text{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}.$$

SOLVING an OPTIMIZATION PROBLEM

Observe that to solve an optimization problem is more as computing a relation, than as computing a function, because we are usually happy with finding one of the optimal solutions or even with a solution quite close to an optimal one.

A feasible solution $\alpha \in \mathcal{M}(x)$ is called optimal for the problem instance x of $\mathcal{P}$ if

$$\text{cost}(\alpha, x) = \text{goal}\{\text{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}.$$

An algorithm $\mathcal{A}$ is said to solve $\mathcal{P}$ if, for any $x \in L$,

- $\mathcal{A}(x) \in \mathcal{M}(x)$;
- $\text{cost}(\mathcal{A}(x), x) = \text{goal}\{\text{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}$

SOLVING an OPTIMIZATION PROBLEM

Observe that to solve an optimization problem is more as computing a relation, than as computing a function, because we are usually happy with finding one of the optimal solutions or even with a solution quite close to an optimal one.

A feasible solution $\alpha \in \mathcal{M}(x)$ is called optimal for the problem instance x of $\mathcal{P}$ if

$$\text{cost}(\alpha, x) = \text{goal}\{\text{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}.$$

An algorithm $\mathcal{A}$ is said to solve $\mathcal{P}$ if, for any $x \in L$,

- $\mathcal{A}(x) \in \mathcal{M}(x)$;
- $\text{cost}(\mathcal{A}(x), x) = \text{goal}\{\text{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}$

If the goal is to find minimum (maximum) we talk about a minimization (maximization) problem.

EXAMPLE - MINIMUM COST HAMILTONIAN CYCLE PROBLEM

EXAMPLE - MINIMUM COST HAMILTONIAN CYCLE PROBLEM

Input: A weighted complete graph (G, c), where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $E \subset V \times V$ and $c : E \to \mathbf{N}$ is a **cost function**.

EXAMPLE - MINIMUM COST HAMILTONIAN CYCLE PROBLEM

Input: A weighted complete graph (G, c), where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $E \subset V \times V$ and $c : E \to \mathbf{N}$ is a **cost function**.

Hamiltonian cycles: For any problem instance (G, c), let $\mathcal{M}(G, c)$ be the set of Hamiltonian cycles of G - each such cycle is represented by a sequence of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$, where $i_1, i_2, \ldots, i_n$ is a permutation of $(1, 2, \ldots, n)$ and for each j, $(v_{i_j}, v_{i_{j+1}})$ is an edge of G

EXAMPLE - MINIMUM COST HAMILTONIAN CYCLE PROBLEM

Input: A weighted complete graph (G, c), where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $E \subset V \times V$ and $c : E \to \mathbf{N}$ is a **cost function**.

Hamiltonian cycles: For any problem instance (G, c), let $\mathcal{M}(G, c)$ be the set of Hamiltonian cycles of G - each such cycle is represented by a sequence of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$, where $i_1, i_2, \ldots, i_n$ is a permutation of $(1, 2, \ldots, n)$ and for each j, $(v_{i_j}, v_{i_{j+1}})$ is an edge of G

Costs of Hamiltonian cycles: For every Hamiltonian cycle $H = (v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}) \in \mathcal{M}(G, c)$

$$\text{cost}((v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}), (G, c)) = \sum_{j=1}^{n} c((v_{i_j}, v_{i_{(j+1) \bmod n}})).$$

EXAMPLE - MINIMUM COST HAMILTONIAN CYCLE PROBLEM

Input: A weighted complete graph (G, c), where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $E \subset V \times V$ and $c : E \to \mathbf{N}$ is a **cost function**.

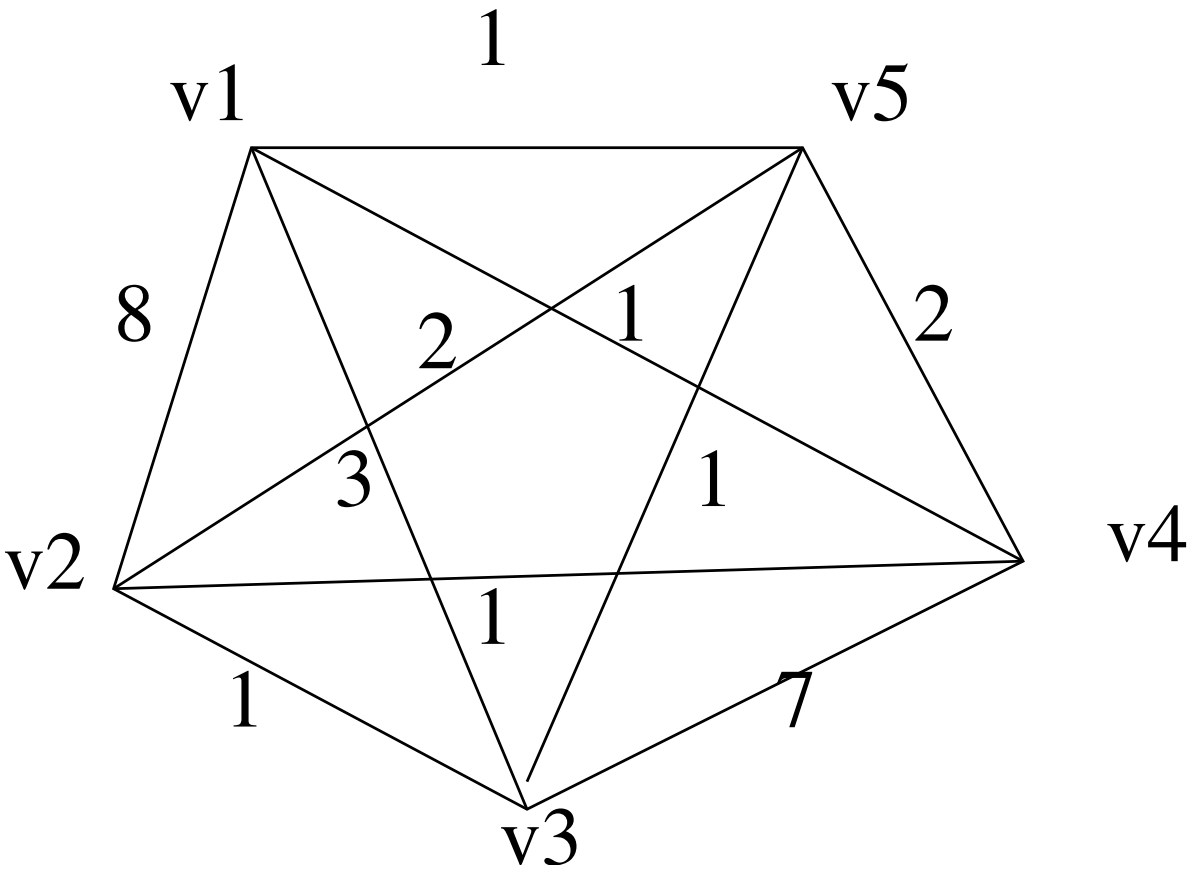
Hamiltonian cycles: For any problem instance (G, c), let $\mathcal{M}(G, c)$ be the set of Hamiltonian cycles of G - each such cycle is represented by a sequence of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$, where $i_1, i_2, \ldots, i_n$ is a permutation of $(1, 2, \ldots, n)$ and for each j, $(v_{i_j}, v_{i_{j+1}})$ is an edge of G

Costs of Hamiltonian cycles: For every Hamiltonian cycle $H = (v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}) \in \mathcal{M}(G, c)$

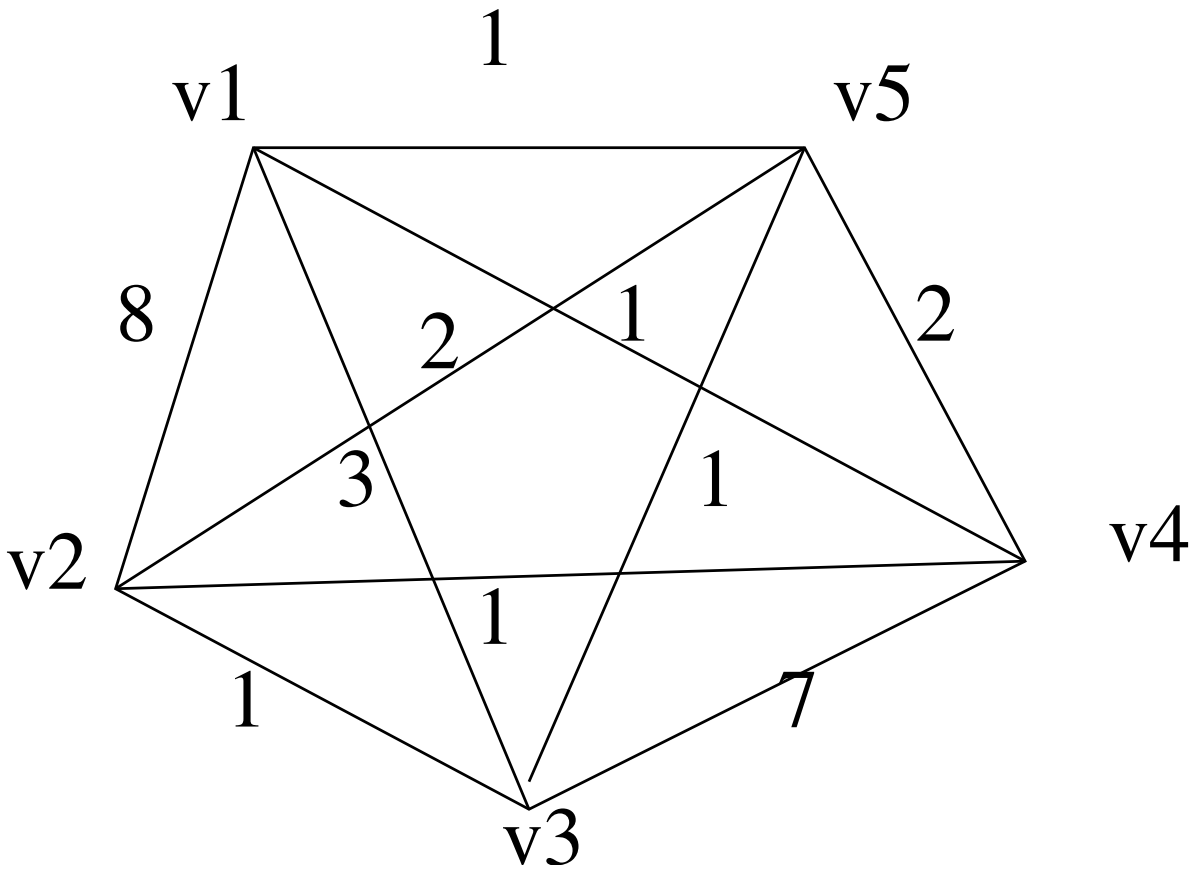
$$\text{cost}((v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}), (G, c)) = \sum_{j=1}^{n} c((v_{i_j}, v_{i_{(j+1) \bmod n}})).$$

Goal: minimum - to find a Hamiltonian cycle with minimum cost.

EXAMPLE

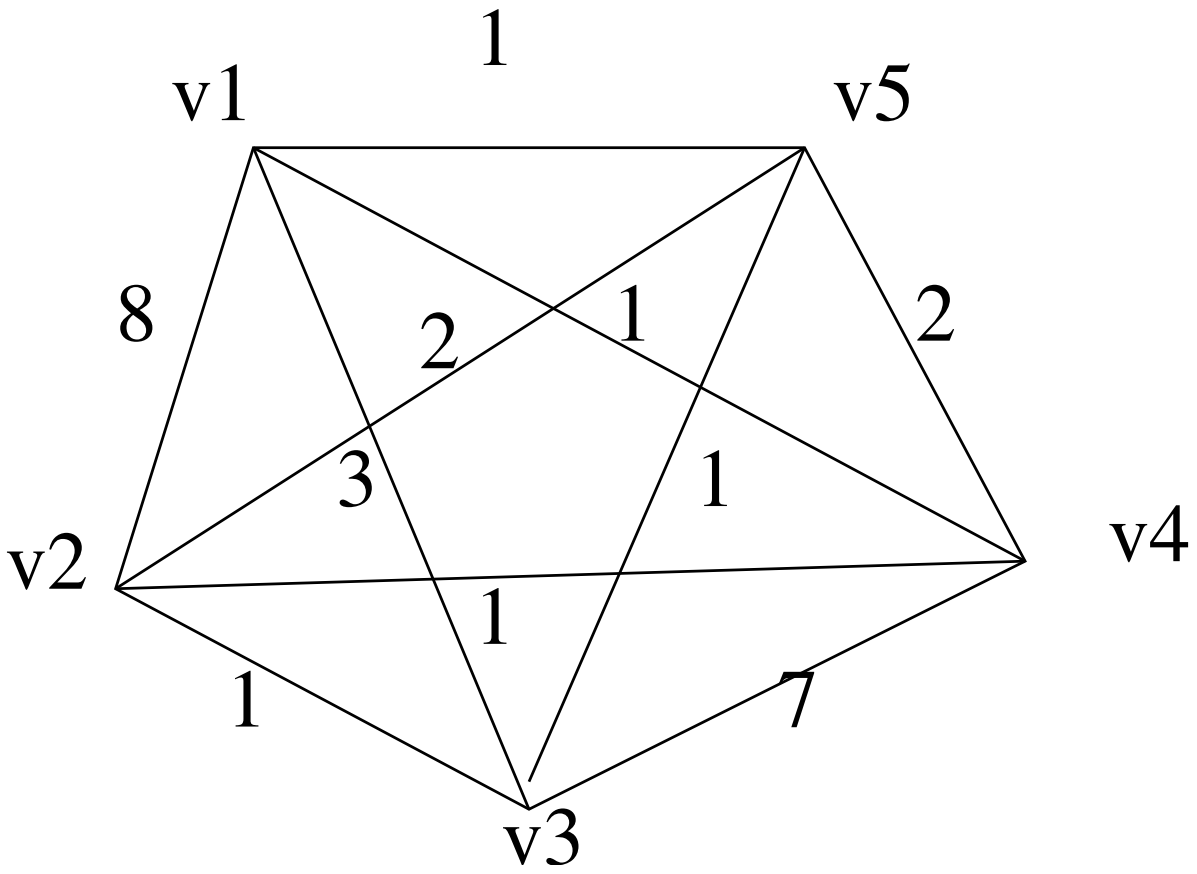


EXAMPLE



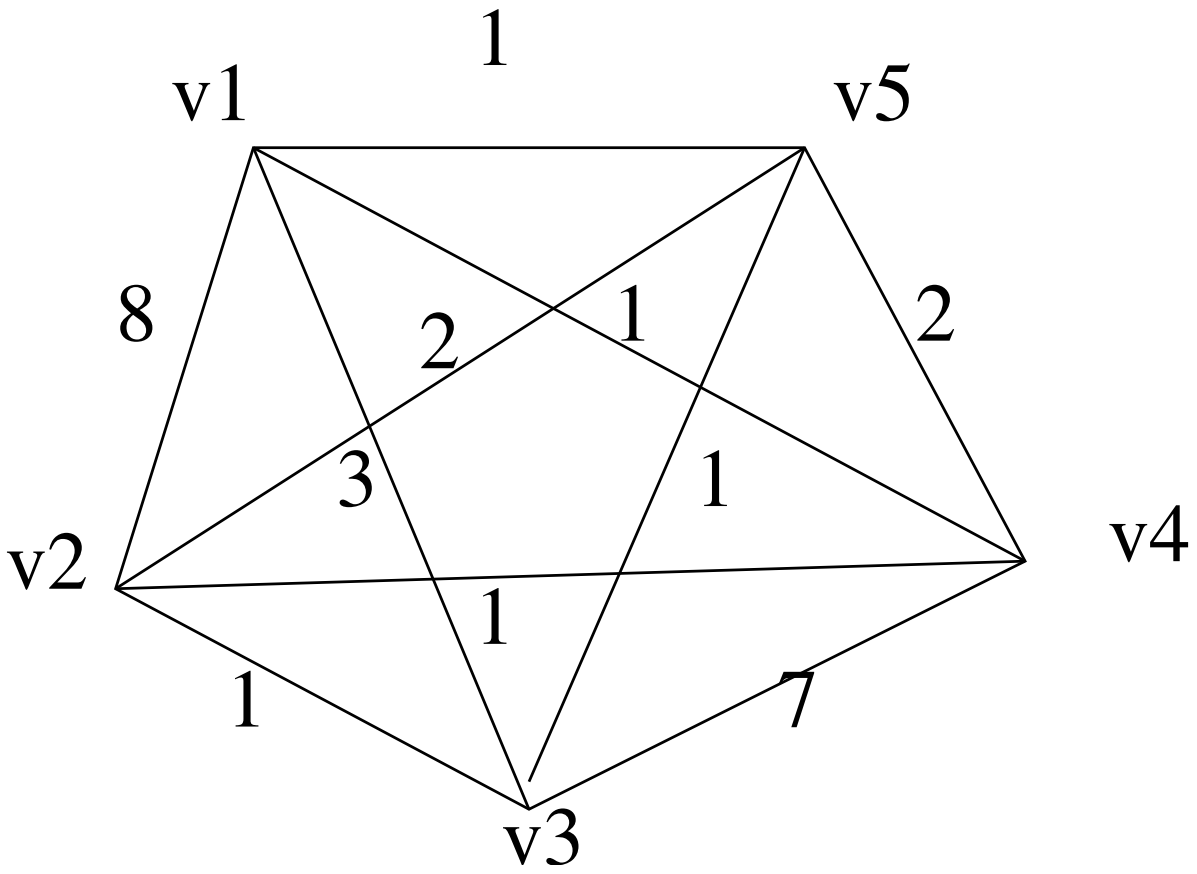
$$\text{cost}((v1, v2, v3, v4, v5, v1), (G, c)) =$$

EXAMPLE



$$\text{cost}((v1, v2, v3, v4, v5, v1), (G, c)) = 19$$

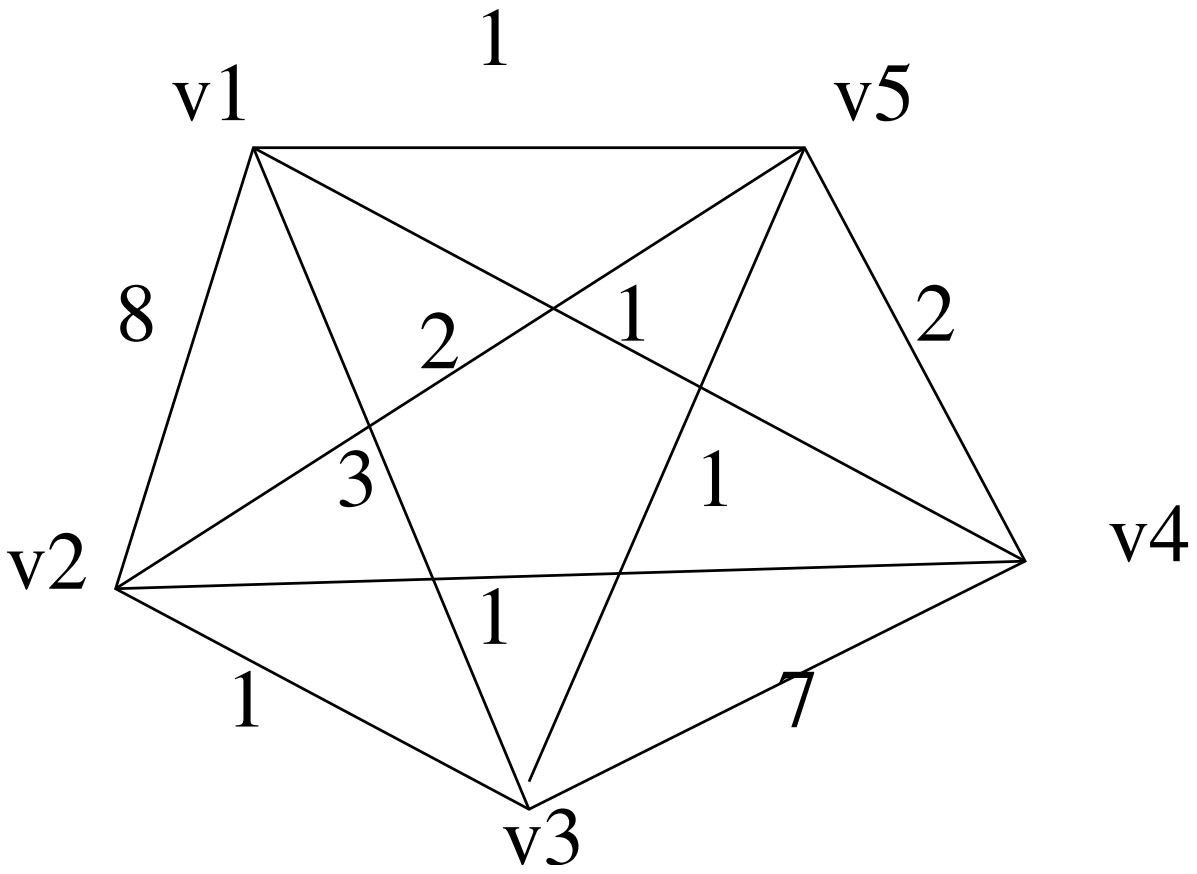
EXAMPLE



$$\text{cost}((v1, v2, v3, v4, v5, v1), (G, c)) = 19$$

$$\text{cost}((v1, v5, v3, v2, v4, v1), (G, c) =$$

EXAMPLE



$$\text{cost}((v1, v2, v3, v4, v5, v1), (G, c)) = 19$$

$$\text{cost}((v1, v5, v3, v2, v4, v1), (G, c) = 5$$

INTEGER LINEAR PROGRAMMING

INTEGER LINEAR PROGRAMMING

Given is a system of linear equations and a linear function over variables of this equations.

INTEGER LINEAR PROGRAMMING

Given is a system of linear equations and a linear function over variables of this equations. The task is to find a solution to the system of equations such that the value of the linear function is minimized.

INTEGER LINEAR PROGRAMMING

Given is a system of linear equations and a linear function over variables of this equations. The task is to find a solution to the system of equations such that the value of the linear function is minimized. More formally:

INTEGER LINEAR PROGRAMMING

Given is a system of linear equations and a linear function over variables of this equations. The task is to find a solution to the system of equations such that the value of the linear function is minimized. More formally:

Input An $m \times n$ matrix and two vectors

$$A = \{a_{ij}\}_{i,j=1}^{i=m,j=n}, \quad b = (b_1, \ldots, b_m)^T, \quad c = (c_1, \ldots, c_n)$$

with integer entries.

INTEGER LINEAR PROGRAMMING

Given is a system of linear equations and a linear function over variables of this equations. The task is to find a solution to the system of equations such that the value of the linear function is minimized. More formally:

Input An $m \times n$ matrix and two vectors

$$A = \{a_{ij}\}_{i,j=1}^{i=m,j=n}, \quad b = (b_1, \ldots, b_m)^T, \quad c = (c_1, \ldots, c_n)$$

with integer entries.

Set of feasible solutions: $\mathcal{M}(A, b) = \{X = (x_1, \ldots, x_n)^T \mid AX = b\}$.

INTEGER LINEAR PROGRAMMING

Given is a system of linear equations and a linear function over variables of this equations. The task is to find a solution to the system of equations such that the value of the linear function is minimized. More formally:

Input An $m \times n$ matrix and two vectors

$$A = \{a_{ij}\}_{i,j=1}^{i=m,j=n}, \quad b = (b_1, \ldots, b_m)^T, \quad c = (c_1, \ldots, c_n)$$

with integer entries.

Set of feasible solutions: $\mathcal{M}(A, b) = \{X = (x_1, \ldots, x_n)^T \mid AX = b\}$.

Cost of a solution: For $X = (x_1, \ldots, x_n) \in \mathcal{M}(A, b)$

$$\text{cost}(X, c)) = c \cdot X = \sum_{i=1}^{n} c_i x_i.$$

Goal: minimum

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 1/2

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 1/2

Many very important optimization problems are **NP**-hard and so we know only exponential time algorithms for finding optimal solutions.

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 1/2

Many very important optimization problems are **NP**-hard and so we know only exponential time algorithms for finding optimal solutions.

New idea is to jump from exponential to polynomial time by weakening the requirements - to be satisfied with almost optimal solutions.

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 1/2

Many very important optimization problems are **NP**-hard and so we know only exponential time algorithms for finding optimal solutions.

New idea is to jump from exponential to polynomial time by weakening the requirements - to be satisfied with almost optimal solutions. To quantize that tries the next definition.

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 1/2

Many very important optimization problems are **NP**-hard and so we know only exponential time algorithms for finding optimal solutions.

New idea is to jump from exponential to polynomial time by weakening the requirements - to be satisfied with almost optimal solutions. To quantize that tries the next definition.

Definition Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. We say that A is a consistent algorithm for $\mathcal{P}$ if, for every $x \in L$, the output $A(x)$ is a feasible solution for x - that is $A(x) \in \mathcal{M}(x)$.
We say that an approximation algorithm A mapping each instance x of an optimization problem P to one of its feasible solutions has the ratio bound $\rho_A(n)$ and the relative error bound $\varepsilon_A(n)$ if

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 1/2

Many very important optimization problems are **NP**-hard and so we know only exponential time algorithms for finding optimal solutions.

New idea is to jump from exponential to polynomial time by weakening the requirements - to be satisfied with almost optimal solutions. To quantize that tries the next definition.

Definition Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. We say that A is a consistent algorithm for $\mathcal{P}$ if, for every $x \in L$, the output $A(x)$ is a feasible solution for x - that is $A(x) \in \mathcal{M}(x)$.
We say that an approximation algorithm A mapping each instance x of an optimization problem P to one of its feasible solutions has the ratio bound $\rho_A(n)$ and the relative error bound $\varepsilon_A(n)$ if

$$\max_{|x|=n} \left\{ \frac{\text{cost}(A(x))}{\text{cost}(\text{Opt}(x))}, \frac{\text{cost}(\text{Opt}(x))}{\text{cost}(A(x))} \right\} \leq \rho_A(n)$$

and

$$\max_{|x|=n} \left\{ \frac{|\text{cost}(A(x)) - \text{cost}(\text{Opt}(x))|}{\max\{\text{cost}(\text{Opt}(x)), \text{cost}(A(x))\}} \right\} \leq \varepsilon_A(n)$$

Both definitions

Both definitions

$$\max_{|x|=n} \left\{ \frac{\text{cost}(A(x))}{\text{cost}(\text{Opt}(x))}, \frac{\text{cost}(\text{Opt}(x))}{\text{cost}(A(x))} \right\} \leq \rho_A(n)$$

and

$$\max_{|x|=n} \left\{ \frac{|\text{cost}(A(x)) - \text{cost}(\text{Opt}(x))|}{\max\{\text{cost}(\text{Opt}(x)), \text{cost}(A(x))\}} \right\} \leq \varepsilon_A(n)$$

are chosen to correspond to our intuition and to apply simultaneously to minimization and maximization problems. Both of these bounds compare an approximation solution with the optimal one, but in two different ways.

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 2/2

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 2/2

For any $\delta > 1$ we say that A is a δ-**approximation algorithm** for $\mathcal{P}$ if, for every integer n, $\rho_A(n) \leq \delta$.

QUANTIFICATION of "ALMOST OPTIMAL SOLUTIONS" 2/2

For any $\delta > 1$ we say that A is a δ-**approximation algorithm** for $\mathcal{P}$ if, for every integer n, $\rho_A(n) \leq \delta$.

The ratio bound is never less than one.
An optimal algorithm has ratio bound 1.
The larger is the best possible ratio bound of an approximation algorithm, the worse is the algorithm.

Approximation algorithms for NP problems

Approximation algorithms for NP problems

Two general problems concerning approximation of **NP**-complete problems are of special interest and importance.

Approximation algorithms for NP problems

Two general problems concerning approximation of **NP**-complete problems are of special interest and importance.

The constant relative error bound problem: Given an **NP**-complete optimization problem P with a cost of solution function c and an $\varepsilon > 0$, does there exist an approximation polynomial time algorithm for P with the relative error bound ε?

Approximation algorithms for NP problems

Two general problems concerning approximation of **NP**-complete problems are of special interest and importance.

The constant relative error bound problem: Given an **NP**-complete optimization problem P with a cost of solution function c and an $\varepsilon > 0$, does there exist an approximation polynomial time algorithm for P with the relative error bound ε?

The approximation scheme problem: Given an **NP**-complete problem P, does there exist for P with a cost of solution function c a polynomial time algorithm for designing, given an $\varepsilon > 0$ and an input instance x, an approximation for P and x with the relative error bound ε?

Approximation thresholds

Approximation thresholds

It is said that an algorithm $\mathcal{A}$ is an ε-approximation algorithm for an optimization problem P if ε is its relative error bound.

Approximation thresholds

It is said that an algorithm $\mathcal{A}$ is an ε-approximation algorithm for an optimization problem P if ε is its relative error bound.

The approximation threshold for P is the greatest lower bound of all $\varepsilon > 0$ such that there is a polynomial time ε-approximation algorithm for P.

Approximation thresholds

It is said that an algorithm $\mathcal{A}$ is an ε-approximation algorithm for an optimization problem P if ε is its relative error bound.

The approximation threshold for P is the greatest lower bound of all $\varepsilon > 0$ such that there is a polynomial time ε-approximation algorithm for P.

It can be shown that **NP**-complete problems can differ very much with respect to their approximation thresholds.

Approximation thresholds

It is said that an algorithm $\mathcal{A}$ is an ε-approximation algorithm for an optimization problem P if ε is its relative error bound.

The approximation threshold for P is the greatest lower bound of all $\varepsilon > 0$ such that there is a polynomial time ε-approximation algorithm for P.

It can be shown that **NP**-complete problems can differ very much with respect to their approximation thresholds.

Note that if an optimization problem P has an approximation threshold 0, this means that a (polynomial time) approximation arbitrarily close to the optimum is possible.

Approximation thresholds

It is said that an algorithm $\mathcal{A}$ is an ε-approximation algorithm for an optimization problem P if ε is its relative error bound.

The approximation threshold for P is the greatest lower bound of all $\varepsilon > 0$ such that there is a polynomial time ε-approximation algorithm for P.

It can be shown that **NP**-complete problems can differ very much with respect to their approximation thresholds.

Note that if an optimization problem P has an approximation threshold 0, this means that a (polynomial time) approximation arbitrarily close to the optimum is possible.

Note also that if P has approximation threshold 1, this means that no universal (polynomial time) approximation method is possible.

Examples

Examples

Example 1 The approximation threshold for the optimization version of the **KNAPSACK PROBLEM** is 0.

Examples

Example 1 The approximation threshold for the optimization version of the **KNAPSACK PROBLEM** is 0.

Example 2 The approximation threshold for the **VERTEX COVER PROBLEM** is $\leq \frac{1}{2}$.

Examples

Example 1 The approximation threshold for the optimization version of the **KNAPSACK PROBLEM** is 0.

Example 2 The approximation threshold for the **VERTEX COVER PROBLEM** is $\leq \frac{1}{2}$.

Example 3 Unless **P** $=$ **NP**, thee approximation threshold for the **TRAVELING SALESMAN PROBLEM** is 1.

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

One of the main goals in the area of discrete optimization is to improve the approximation ratio.

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

One of the main goals in the area of discrete optimization is to improve the approximation ratio. One tries to design randomized approximation algorithms that produce feasible solutions whose cost (quality) is not very far from the optimal cost with high probability.

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

One of the main goals in the area of discrete optimization is to improve the approximation ratio. One tries to design randomized approximation algorithms that produce feasible solutions whose cost (quality) is not very far from the optimal cost with high probability. In the analysis of randomized algorithms we consider therefore the approximation ratio as a random variable and the aim is then either

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

One of the main goals in the area of discrete optimization is to improve the approximation ratio. One tries to design randomized approximation algorithms that produce feasible solutions whose cost (quality) is not very far from the optimal cost with high probability. In the analysis of randomized algorithms we consider therefore the approximation ratio as a random variable and the aim is then either

1. to estimate the expected value, E(Ratio), or

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

One of the main goals in the area of discrete optimization is to improve the approximation ratio. One tries to design randomized approximation algorithms that produce feasible solutions whose cost (quality) is not very far from the optimal cost with high probability. In the analysis of randomized algorithms we consider therefore the approximation ratio as a random variable and the aim is then either

1. to estimate the expected value, E(Ratio), or
2. to guarantee that a certain approximation ratio is achieved with probability at least $\frac{1}{2}$.

WHY TO APPLY RANDOMIZATION in DISCRETE OPTIMIZATIONS

One of the main goals in the area of discrete optimization is to improve the approximation ratio. One tries to design randomized approximation algorithms that produce feasible solutions whose cost (quality) is not very far from the optimal cost with high probability. In the analysis of randomized algorithms we consider therefore the approximation ratio as a random variable and the aim is then either

1. to estimate the expected value, E(Ratio), or
2. to guarantee that a certain approximation ratio is achieved with probability at least $\frac{1}{2}$.

These two different aims lead to two ways randomized approximation algorithms are defined.

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized E$[\delta]$-approximation algorithm** for $\mathcal{P}$ if

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized E$[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized E$[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\mathbf{E}[\text{Ratio}_A(x) \leq \delta] \geq \frac{1}{2}$.

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized $\mathbf{E}[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\mathbf{E}[\text{Ratio}_A(x) \leq \delta] \geq \frac{1}{2}$.

for every $x \in L$.

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized $\mathbf{E}[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\mathbf{E}[\text{Ratio}_A(x) \leq \delta] \geq \frac{1}{2}$.

for every $x \in L$.

Definition Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem.For any $\delta > 1$, a randomized algorithm A is called a **randomized δ-approximation algorithm** for $\mathcal{P}$ if

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized $\mathbf{E}[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\mathbf{E}[\text{Ratio}_A(x) \leq \delta] \geq \frac{1}{2}$.

for every $x \in L$.

Definition Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem.For any $\delta > 1$, a randomized algorithm A is called a **randomized δ-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized $\mathbf{E}[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\mathbf{E}[\text{Ratio}_A(x) \leq \delta] \geq \frac{1}{2}$.

for every $x \in L$.

Definition Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem.For any $\delta > 1$, a randomized algorithm A is called a **randomized δ-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\text{Prob}(\text{Ratio}_A(x) \leq \delta) \geq \frac{1}{2}$.

Definition 1 Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For any $\delta > 1$, a randomized algorithm A is called a **randomized $\mathbf{E}[\delta]$-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\mathbf{E}[\text{Ratio}_A(x) \leq \delta] \geq \frac{1}{2}$.

for every $x \in L$.

Definition Let $\mathcal{P} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem.For any $\delta > 1$, a randomized algorithm A is called a **randomized δ-approximation algorithm** for $\mathcal{P}$ if

1. $\text{Prob}(A(x) \in \mathcal{M}(x)) = 1$, and
2. $\text{Prob}(\text{Ratio}_A(x) \leq \delta) \geq \frac{1}{2}$.

for every $x \in L$.

Online algorithms

Online algorithms

Another area in which randomization plays important role are online algorithms.

Online algorithms

Another area in which randomization plays important role are online algorithms.

In practice the following problems are often of importance.

Online algorithms

Another area in which randomization plays important role are online algorithms.

In practice the following problems are often of importance. One always obtains only a part of the input that has to be processed immediately.

Online algorithms

Another area in which randomization plays important role are online algorithms.

In practice the following problems are often of importance. One always obtains only a part of the input that has to be processed immediately. Once this is done one obtains another part of the input and has to process it again immediately, and so on - the input can be infinitely long.

Online algorithms

Another area in which randomization plays important role are online algorithms.

In practice the following problems are often of importance. One always obtains only a part of the input that has to be processed immediately. Once this is done one obtains another part of the input and has to process it again immediately, and so on - the input can be infinitely long. Such problems are called online problems and algorithms to solve them are called online algorithms.

Online algorithms

Another area in which randomization plays important role are online algorithms.

In practice the following problems are often of importance. One always obtains only a part of the input that has to be processed immediately. Once this is done one obtains another part of the input and has to process it again immediately, and so on - the input can be infinitely long. Such problems are called online problems and algorithms to solve them are called online algorithms.

Example Scheduling problem - immediate assigning of resources for requests coming one after another.

Online algorithms

Another area in which randomization plays important role are online algorithms.

In practice the following problems are often of importance. One always obtains only a part of the input that has to be processed immediately. Once this is done one obtains another part of the input and has to process it again immediately, and so on - the input can be infinitely long. Such problems are called online problems and algorithms to solve them are called online algorithms.

Example Scheduling problem - immediate assigning of resources for requests coming one after another.

Key question. How good can an online algorithm (that does not know the future) be in comparison to an algorithm that knows the whole input (the future) from the beginning?

Evaluation of online algorithms

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1]

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.
2. $A(x) \in M(x)$, i.e. A always computes a feasible solution.

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.
2. $A(x) \in M(x)$, i.e. A always computes a feasible solution.
3. For all $i \in \{1, \dots, n\}$, $A(x_1 x_2 \dots x_i)$ is a part of $A(x)$, i.e. the decisions made for the prefix $x_1 x_2 \dots x_i$ of x cannot be changed any more.

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.
2. $A(x) \in M(x)$, i.e. A always computes a feasible solution.
3. For all $i \in \{1, \dots, n\}$, $A(x_1 x_2 \dots x_i)$ is a part of $A(x)$, i.e. the decisions made for the prefix $x_1 x_2 \dots x_i$ of x cannot be changed any more.

For every input $x \in L$, the **competitive ratio** $comp_A(x)$ **of** A **on** x is the number

$$\mathbf{comp_A(x)} = \max\left\{\frac{\text{Opt}_P(x)}{\text{cost}_A(x)}, \frac{\text{cost}_A(x)}{\text{Opt}_P(x)}\right\}$$

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.
2. $A(x) \in M(x)$, i.e. A always computes a feasible solution.
3. For all $i \in \{1, \dots, n\}$, $A(x_1 x_2 \dots x_i)$ is a part of $A(x)$, i.e. the decisions made for the prefix $x_1 x_2 \dots x_i$ of x cannot be changed any more.

For every input $x \in L$, the **competitive ratio** $comp_A(x)$ **of** A **on** x is the number

$$\mathbf{comp_A(x)} = \max\left\{\frac{\text{Opt}_P(x)}{\text{cost}_A(x)}, \frac{\text{cost}_A(x)}{\text{Opt}_P(x)}\right\}$$

where $\text{Opt}_P(x)$ denotes the cost of an optimal solution for the instance x of the problem P.

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.
2. $A(x) \in M(x)$, i.e. A always computes a feasible solution.
3. For all $i \in \{1, \dots, n\}$, $A(x_1 x_2 \dots x_i)$ is a part of $A(x)$, i.e. the decisions made for the prefix $x_1 x_2 \dots x_i$ of x cannot be changed any more.

For every input $x \in L$, the **competitive ratio** $comp_A(x)$ **of** A **on** x is the number

$$\mathbf{comp_A(x)} = \max\left\{\frac{\text{Opt}_P(x)}{\text{cost}_A(x)}, \frac{\text{cost}_A(x)}{\text{Opt}_P(x)}\right\}$$

where $\text{Opt}_P(x)$ denotes the cost of an optimal solution for the instance x of the problem P.

Let $\delta \geq 1$. We say that A is a δ-**competitive algorithm** for P if $\text{comp}_A(x) \leq \delta$ for all $x \in L$.

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

Evaluation of online algorithms

Let $P == (\Sigma_i, \Sigma_0, L, M, \text{cost}, \text{goal})$ be an optimization problem that can be viewed as an online problem[1] An algorithm A is an online algorithm for P if, for every input $x = x_1 x_2 \dots x_n \in L$ the following conditions are satisfied:

1. For all $i \in \{1, \dots, n\}$ $x_1 x_2 \dots x_i$ is a feasible input.
2. $A(x) \in M(x)$, i.e. A always computes a feasible solution.
3. For all $i \in \{1, \dots, n\}$, $A(x_1 x_2 \dots x_i)$ is a part of $A(x)$, i.e. the decisions made for the prefix $x_1 x_2 \dots x_i$ of x cannot be changed any more.

For every input $x \in L$, the **competitive ratio** $comp_A(x)$ **of** A **on** x is the number

$$\mathbf{comp_A(x)} = \max\left\{\frac{\text{Opt}_P(x)}{\text{cost}_A(x)}, \frac{\text{cost}_A(x)}{\text{Opt}_P(x)}\right\}$$

where $\text{Opt}_P(x)$ denotes the cost of an optimal solution for the instance x of the problem P.

Let $\delta \geq 1$. We say that A is a δ-**competitive algorithm** for P if $\text{comp}_A(x) \leq \delta$ for all $x \in L$.

Let $\delta \geq 1$ be a real. We say that an online problem P is δ-**hard** if there does not exist any d-competitive online algorithm for P with $d < \delta$.

[1] An optimization problem can be viewed as an online problem when each prefix y of every input x can be viewed also as a problem instance, and one is required to provide a solution for y that has to remain unchanged as a part of the solution for the whole input x.

RANDOMNESS EXTRACTORS

RANDOMNESS EXTRACTORS

Extractors are algorithms that produce from any long and weakly-random bit-string a shorter, but more random, bit-string.

RANDOMNESS EXTRACTORS

Extractors are algorithms that produce from any long and weakly-random bit-string a shorter, but more random, bit-string.

In other words, an extractor is a mapping which, when applied to high-entropy source generates a shorter yet uniformly distributed output.

RANDOMNESS EXTRACTORS

Extractors are algorithms that produce from any long and weakly-random bit-string a shorter, but more random, bit-string.

In other words, an extractor is a mapping which, when applied to high-entropy source generates a shorter yet uniformly distributed output.

In a more general approach an extractor is an algorithm that converts a long weakly random source and a truly random short seed into a uniformly distributed random output (that is longer than the seed and shorter than the source).

RANDOMNESS EXTRACTORS

Extractors are algorithms that produce from any long and weakly-random bit-string a shorter, but more random, bit-string.

In other words, an extractor is a mapping which, when applied to high-entropy source generates a shorter yet uniformly distributed output.

In a more general approach an extractor is an algorithm that converts a long weakly random source and a truly random short seed into a uniformly distributed random output (that is longer than the seed and shorter than the source).

An extractor is a certain kind of pseudorandom generator.

RANDOMNESS EXTRACTORS

Extractors are algorithms that produce from any long and weakly-random bit-string a shorter, but more random, bit-string.

In other words, an extractor is a mapping which, when applied to high-entropy source generates a shorter yet uniformly distributed output.

In a more general approach an extractor is an algorithm that converts a long weakly random source and a truly random short seed into a uniformly distributed random output (that is longer than the seed and shorter than the source).

An extractor is a certain kind of pseudorandom generator.

No extractor is currently know that has been proven to work when applied to any type of high-entropy source.

von NEUMANN EXTRACTOR

von NEUMANN EXTRACTOR

It is an extractor that keeps taking successive pairs of consecutive bits (non-overlapping) from the input stream and if the two bits are the same, no output is generated; if they are different the first of them is outputted.

von NEUMANN EXTRACTOR

It is an extractor that keeps taking successive pairs of consecutive bits (non-overlapping) from the input stream and if the two bits are the same, no output is generated; if they are different the first of them is outputted.

For example the input sequence

$$10111100001111000011100110010101$$

is transformed to the sequence

$$1101000$$

von NEUMANN EXTRACTOR

It is an extractor that keeps taking successive pairs of consecutive bits (non-overlapping) from the input stream and if the two bits are the same, no output is generated; if they are different the first of them is outputted.

For example the input sequence

$$10111100001111000011100110010101$$

is transformed to the sequence

$$1101000$$

The von Neumann extractor can be shown to produce a uniform output, even if the distribution of the input bits is not uniform, so long as each bit has the same probability of being one and there is no correlation between successive bits.

EXTRACTORS - FORMAL DEFINITION

EXTRACTORS - FORMAL DEFINITION

Definition A (k, ε)-extractor Ext is a mapping

$$\text{Ext} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$$

such that for every distribution X on $\{0,1\}^n$ with $H_\infty(X) \geq k$ and the seed s, the distribution $\text{Ext}(X, s)$ is ε-close to the uniform distribution on $\{0,1\}^m$.

EXTRACTORS - FORMAL DEFINITION

Definition A (k, ε)-extractor Ext is a mapping

$$\text{Ext} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$$

such that for every distribution X on $\{0,1\}^n$ with $H_\infty(X) \geq k$ and the seed s, the distribution $\text{Ext}(X, s)$ is ε-close to the uniform distribution on $\{0,1\}^m$.

The aim is to have $n > m$ and $d << m$.

EXTRACTORS - FORMAL DEFINITION

Definition A (k, ε)-extractor Ext is a mapping

$$\text{Ext} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$$

such that for every distribution X on $\{0,1\}^n$ with $H_\infty(X) \geq k$ and the seed s, the distribution $\text{Ext}(X, s)$ is ε-close to the uniform distribution on $\{0,1\}^m$.

The aim is to have $n > m$ and $d << m$.

By a probabilistic method to be discussed later one can show that there exists a (k, ε) extractor for many k and ε.

Note H_∞ stands for so-called *min-entropy*, which is a measure of the amount of randomness in the worst case.